Drug Design in Parallel

CSinParallel Project

March 04, 2014

CONTENTS

This document contains several parallel programming solutions to the drug design exemplar using alternative parallel and distributed computing (PDC) technologies. The last chapter provides a discussion of the performance implications of the solutions and the parallel design patterns used in them.

CHAPTER

ONE

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 (read more).

1.1 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; the full sequential implementation is available at `dd_serial.cpp`.

```
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.

1.2 Further 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

1.3 Questions for Exploration

- Compare the performance of `dd_serial.cpp` with `dd_omp.cpp` on a multicore computer using the same values for `max_ligand` and `nligands`. Do you observe speedup for the parallel version?
- Our development system has four cores, and `nthreads=4` was used for one of our test runs. We found that `dd_omp.cpp` performed about *three* times as fast as `dd_serial.cpp` for the same values of `max_ligand` and `nligands`. Can you explain why it didn't perform four times as fast?
- Use the command-line arguments to experiment with varying the number of OpenMP threads in an invocation of `dd_omp.cpp`, while holding `max_ligand` and `nligands` unchanged. On a multi-core system, we hope for better performance when more threads are used. Do you observe such performance improvement when you time the execution? What happens when the number of threads exceeds the number of cores (or hyperthreads) on your system? Explain as much as you can about the timing results you observe when you vary the number of threads.
- You may notice that `dd_omp.cpp` computes the same maximal score and identifies the same ligands as `dd_serial.cpp` that produce that score, but if more than one ligand yields the maximal score, the *order* of those maximal-scoring ligands may differ between the two versions. Can you explain why?
- Our sequential program `dd_serial.cpp` always produces the same results for given values of the `max_ligand`, `nligands`, and `protein` command-line arguments. This is because we use the default random-number seed in our code. Because of this consistency, we can describe the sequential version as being a *deterministic* computation. Is `dd_omp.cpp` a deterministic computation? Explain your answer, and/or state what more you need to know in order to answer this question.
- If you have *more realistic algorithms for docking and/or more realistic data for ligands and proteins*, modify the program `dd_omp.cpp` to incorporate those elements, and compare the results from your modified program to results obtained by other means (other software, wet-lab results, etc.). How does the performance of your modified OpenMP version compare to what you observed from your modified sequential version?

- Whereas our serial implementation used a queue data structure for `tasks`, this implementation uses a vector data structure, and parallelizes the "map" stage using OpenMP's `omp parallel for` pragma. This suffices for our simplified example, because we generate all ligands before processing any of them. However, some computations require a task queue, since processing some tasks may generate others (not out of the question for drug design, since high-scoring ligands might lead one to consider similar ligands in search of even higher scores). **Challenge problem:** Modify `dd_omp.cpp` to use a task *queue* instead of a task vector.

  Note:

  – Use a thread-safe queue data structure for `tasks`, such as `tbb::concurrent_queue` or `tbb::concurrent_bounded_queue`, because multiple threads may attempt to modify the queue at the same time.

  – Instead of `omp parallel for`, use OpenMP 3.0 tasks. You can parallelize a `while` loop that moves through the task queue using `omp parallel` to enclose that loop.

  – Depending on your algorithm, it may help to use "sentinel" values, as described in Chapter 8 of this book or as used by the Boost threads implementation in the next page.

CHAPTER

TWO

BOOST THREADS SOLUTION

In the OpenMP implementation above, the OpenMP runtime system implicitly creates and manages threads for us. The implementation `dd_boost.cpp` parallelizes the computationally expensive `Map()` stage by using Boost threads (precursors of 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`, a task queue produced by `Generate_tasks()`.

We will examine the Boost threads implementation `dd_boost.cpp` by comparing it to the sequential implementation `dd_serial.cpp`.

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

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

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

In this snippet from the Boost 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.

2.1 Further 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 `dd_omp.cpp`, the Boost 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.

2.2 Questions for exploration

- Compile and run `dd_boost.cpp`, and compare its performance to `dd_serial.cpp` and to other parallel implementations.
- *Concurrent task queue:* consider the "map" stage in our sequential implementation `dd_serial.cpp`, which uses an STL container instead of a TBB container for the task queue `tasks`:

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

  Note:

  - TBB container classes `tbb::concurrent_queue` and `tbb::concurrent_bounded_queue` do not provide a method `front()`. Instead, they provide a method `try_pop()` (with one argument). It works as follows: if the queue is empty, it returns immediately (non-blocking) without making any changes. If the queue is non-empty, it removes the first element from the queue and assigns it to the argument. This accomplishes the work of an STL `queue`'s `front()` and `pop()` methods in a single operation. Describe a parallel computing scenario in which a single (atomic) operation `try_pop()` is preferable to separate operations `front()` and `pop()`, and explain why we should prefer it.
  - Given that we choose a TBB queue container for the type of `tasks`, would it be safe to have multiple threads execute the following code (which more closely mirrors our sequential operation)?

```
string lig;
while (!tasks.empty()) {
        tasks.try_pop(lig);
        Map(lig, pairs);
}
```

If it *is* safe, explain how you know it is so. If something can go wrong with this code, describe a scenario in which it fails to behave correctly.

- The purpose of `SENTINEL` in our Boost threads implementation is to insure that every (non-`SENTINEL`) element in the task queue `tasks` is processed by some thread, and that all threads terminate (return from `do_Maps()`) when no more (non-`SENTINEL`) elements are available. Verify that this goal is achieved in `dd_boost.cpp`, or describe a scenario in which the goal fails.
- Revise `dd_boost.cpp` to use a `tbb::concurrent_queue` container instead of a `tbb::concurrent_bounded_queue` container for the task queue `tasks`.

Note:

- `tbb::concurrent_queue` does not provide the blocking method `pop()` used in `dd_boost.cpp`, so some other synchronization strategy will be required.
- However, in our simplified problem, the task queue `tasks` doesn't change during the "map" stage, so threads may finish once `tasks` becomes empty.
- Be sure to understand the concurrent task queue exercise above (italicized) before attempting this exercise.
- Is a `SENTINEL` value needed for your solution?

- For further ideas, see exercises for other parallel implementations.

CHAPTER

THREE

GO SOLUTION

Google's Go language makes it possible to program with implicitly launched threads, and its channel feature enables simplified thread-safe processing of shared data.

We will compare the "map" stage in the Go implementation `dd_go.go` to the "map" stage in the Boost thread code `dd_boost.cpp`. The segment of `main()` in `dd_go.go` that implements the "map" stage appears below.

```
pairs := make(chan Pair, 1024)
for i := 0; i < *nCPU; i++ {
go func() {
                p := []byte(*protein)
                for l := range ligands {
                        pairs <- Pair{score(l, p), l}
                }
        }()
}
```

Instead of a vector of `Pair` as in `dd_boost.cpp` , the Go implementation creates a *channel* object called `pairs` for communicating `Pair` objects through message passing. The "map" stage will send `Pairs` into the channel `pairs`, and the sorting stage will receive those `Pairs` from that same channel. In effect, channel `pairs` behaves like a queue, in which the send operation (`<-`) functions like `push_back` and the receive operation (also `<-`, but with the channel on the right side; not shown in the snippet above) acts like `pop`.

The Boost implementation allocated an array `pool` of threads, then had each thread call `do_Map()` in order to carry out that thread's work in the "map" stage. The following code from `dd_boost.cpp` accomplished these operations.

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

Instead of explicitly constructing and storing threads, the Go implementation uses the construct

```
go func() {
                        ...
        }()
```

This `go` statement launches threads that each execute an (anonymous) function to do their work, i.e., carry out the (omitted) instructions indicated by the ellipses ... (in essence, these instructions carry out the work corresponding to `do_Maps`). Note that we could also have defined that as a function `foo()` elsewhere and called it this way (i.e., `go foo()`), but Go is able to employ anonymous functions because it is garbage-collected.

In the Boost threads implementation, the threads must retrieve ligand values repeatedly from a queue `ligands` and then append the retrieved ligand and its score to the vector `pairs`. The methods `do_Maps()` and `Map()` in our Boost implementation accomplish these steps; their code could be combined into something like this.

```
string lig;
tasks.pop(lig);
while (lig != SENTINEL) {
        Pair p(Help::score(ligand.c_str(), protein.c_str()), ligand);
        pairs.push_back(p);
        tasks.pop(lig);
}
tasks.push(SENTINEL);  // restore end marker for another thread
```

In comparison, the goroutines (threads) in the Go implementation carry out the following code.

```
p := []byte(*protein)
        for l := range ligands {
                pairs <- Pair{score(l, p), l}
        }
```

Here, a goroutine obtains its ligand work tasks from a channel `ligands` (created and filled during the "task generation" stage), similarly to the work queue `tasks` in the Boost threads implementation. Also, that ligand and its score are sent to the channel `pairs` discussed above.

3.1 Further Notes

- The use of Go's channel feature made some key parts of the Go code more concise, as seen above. For example, highlighted sections above show that we needed fewer lines of (arguably) less complex code to process a ligand and produce a `Pair` in the Go code than in the Boost threads code. Also, the Go runtime manages thread creation implicitly, somewhat like OpenMP, whereas we must allocate and manage Boost threads explicitly.
- Using channels also simplified the synchronization logic in our Go implementation.
  - We used (thread-safe) Go channels in place of the task queue `tasks` and the vector of Pair `pairs` to manage the flow of our data. Reasoning with the send and receive operations on channels is at least as easy as reasoning about queue and vector operations.
  - The Boost implementation used TBB `concurrent_bounded_queue` instead of `concurrent_queue` because of the availability of a blocking `pop()` operation, so that one could modify `dd_boost.cpp` to include dynamic ligand generation in a straightforward and correct way, and used a value `SENTINEL` to detect when ligands were actually exhausted. Go channels provide these features in a simpler and readily understood way.
- Just after the "map" stage, the Go implementation stores all Pairs in the channel `pairs` into an array for sorting. We cannot store into that array directly during the parallel "map" stage, since that array is not thread-safe.

3.2 Questions for exploration

- Compile and run `dd_go.go`, and compare its performance to `dd_serial.cpp` and to other parallel implementations.

3.2.1 Go Solution

Google's Go language makes it possible to program with implicitly launched threads, and its channel feature enables simplified thread-safe processing of shared data.

We will compare the "map" stage in the Go implementation `dd_go.go` to the "map" stage in the Boost thread code `dd_boost.cpp`. The segment of `main()` in `dd_go.go` that implements the "map" stage appears below.

```
pairs := make(chan Pair, 1024)
for i := 0; i < *nCPU; i++ {
go func() {
                p := []byte(*protein)
                for l := range ligands {
                        pairs <- Pair{score(l, p), l}
                }
        }()
}
```

Instead of a vector of `Pair` as in `dd_boost.cpp` , the Go implementation creates a *channel* object called `pairs` for communicating `Pair` objects through message passing. The "map" stage will send `Pairs` into the channel `pairs`, and the sorting stage will receive those `Pairs` from that same channel. In effect, channel `pairs` behaves like a queue, in which the send operation (`<-`) functions like `push_back` and the receive operation (also `<-`, but with the channel on the right side; not shown in the snippet above) acts like `pop`.

The Boost implementation allocated an array `pool` of threads, then had each thread call `do_Map()` in order to carry out that thread's work in the "map" stage. The following code from `dd_boost.cpp` accomplished these operations.

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

Instead of explicitly constructing and storing threads, the Go implementation uses the construct

```
go func() {
                        ...
        }()
```

This `go` statement launches threads that each execute an (anonymous) function to do their work, i.e., carry out the (omitted) instructions indicated by the ellipses . . . (in essence, these instructions carry out the work corresponding to `do_Maps`). Note that we could also have defined that as a function `foo()` elsewhere and called it this way (i.e., `go foo()`), but Go is able to employ anonymous functions because it is garbage-collected.

In the Boost threads implementation, the threads must retrieve ligand values repeatedly from a queue `ligands` and then append the retrieved ligand and its score to the vector `pairs`. The methods `do_Maps()` and `Map()` in our Boost implementation accomplish these steps; their code could be combined into something like this.

```
string lig;
tasks.pop(lig);
while (lig != SENTINEL) {
        Pair p(Help::score(ligand.c_str(), protein.c_str()), ligand);
        pairs.push_back(p);
        tasks.pop(lig);
}
tasks.push(SENTINEL);  // restore end marker for another thread
```

In comparison, the goroutines (threads) in the Go implementation carry out the following code.

```
p := []byte(*protein)
        for l := range ligands {
                pairs <- Pair{score(l, p), l}
        }
```

Here, a goroutine obtains its ligand work tasks from a channel `ligands` (created and filled during the "task generation" stage), similarly to the work queue `tasks` in the Boost threads implementation. Also, that ligand and its score are sent to the channel `pairs` discussed above.

3.3 Further Notes

- The use of Go's channel feature made some key parts of the Go code more concise, as seen above. For example, highlighted sections above show that we needed fewer lines of (arguably) less complex code to process a ligand and produce a `Pair` in the Go code than in the Boost threads code. Also, the Go runtime manages thread creation implicitly, somewhat like OpenMP, whereas we must allocate and manage Boost threads explicitly.
- Using channels also simplified the synchronization logic in our Go implementation.
  - We used (thread-safe) Go channels in place of the task queue `tasks` and the vector of Pair `pairs` to manage the flow of our data. Reasoning with the send and receive operations on channels is at least as easy as reasoning about queue and vector operations.
  - The Boost implementation used TBB `concurrent_bounded_queue` instead of `concurrent_queue` because of the availability of a blocking `pop()` operation, so that one could modify `dd_boost.cpp` to include dynamic ligand generation in a straightforward and correct way, and used a value `SENTINEL` to detect when ligands were actually exhausted. Go channels provide these features in a simpler and readily understood way.
- Just after the "map" stage, the Go implementation stores all Pairs in the channel `pairs` into an array for sorting. We cannot store into that array directly during the parallel "map" stage, since that array is not thread-safe.

3.4 Questions for exploration

- Compile and run `dd_go.go`, and compare its performance to `dd_serial.cpp` and to other parallel implementations.
- For further ideas, see exercises for other parallel implementations.

CHAPTER

FOUR

HADOOP SOLUTION

Hadoop is an open-source framework for data-intensive scalable map-reduce computation. Originally developed by Yahoo! engineers and now an Apache project, Hadoop supports petascale computations in a reasonable amount of time (given sufficiently large cluster resources), and is used in numerous production web-service enterprises. The code `dd_hadoop.java` implements a solution to our problem for the Hadoop map-reduce framework, which is capable of data-intensive scalable computing.

In our previous examples, we have modified the coding of a map-reduce framework represented by the C++ method `MR::run()` in order to create implementations with various parallelization technologies. Hadoop provides a powerful implementation of such a framework, with optimizations for large-scale data, adaptive scheduling of tasks, automated recovery from failures (which will likely occur when using many nodes for lengthy computations), and an extensive system for reusable configuration of jobs. To use Hadoop, one needs only provide `Map()`, `Reduce()`, configuration options, and the desired data. This framework-based strategy makes it convenient for Hadoop programmers to create and launch effective, scalably large computations.

Therefore, we will compare definitions of `Map()` and `Reduce()` found in the serial implementation `dd_serial.cpp` to the corresponding definitions in a Hadoop implementation `dd_hadoop.java`. The serial implementations for our simplified problem are quite simple:

```
void MR::Map(const string &ligand, vector<Pair> &pairs) {
        Pair p(Help::score(ligand.c_str(), protein.c_str()), ligand);
        pairs.push_back(p);
}

int MR::Reduce(int key, const vector<Pair> &pairs, int index, string &values) {
        while (index < pairs.size() && pairs[index].key == key) {
                values += pairs[index++].val + " ";
        }
        return index;
}
```

Here, `Map()` has two arguments, a ligand to compare to the target protein and an STL vector `pairs` of key-value pairs. A call to `Map()` appends a pair consisting of that ligand's score (as key) and that ligand itself (as value) to the vector `pairs`. Our `Reduce()` function extracts all the key-value pairs from the (now sorted) vector `pairs` having a given key (i.e., score). It then appends a string consisting of all those values (i.e., ligands) to an array `values`. The argument `index` and the return value are used by `MR::run()` in order to manage progress through the vector `pairs`(our multi-threaded implementations have identical `Map()` and `Reduce()` methods, except that a thread-safe vector type is used for `pairs`). In brief, `Map()` receives `ligand` values and produces pairs, and `Reduce()` receives pairs and produces consolidated results in `values`.

In Hadoop, we define the "map" and "reduce" operations as Java methods `Map.map()` and `Reduce.reduce()`. Here are definitions of those methods from :download: *dd_hadoop.java <code/dd_hadoop.java>*:

```
public void map(LongWritable key, Text value, OutputCollector<IntWritable, Text> output, Reporte
                throws IOException {
        String ligand = value.toString();
        output.collect(new IntWritable(score(ligand, protein)), value);
}

                ...

public void reduce(IntWritable key, Iterator<Text> values, OutputCollector<IntWritable, Text> ou
                throws IOException {
        String result = new String("");
        while (values.hasNext()) {
                result += values.next().toString() + " ";
        }
        output.collect(key, new Text(result));
}
```

In brief, our Hadoop implementation's `map()` receives a key and a value, and produces pairs to the `OutputCollector` argument `output`, and `reduce()` receives a key and an iterator of values and produces consolidated results in an `OutputCollector` argument (also named `output`). In Hadoop, the values from key-value pairs sent to a particular call of `reduce()` are provided in an *iterator* rather than a vector or array, since there may be too many values to hold in memory with very large scale data. Likewise, the `OutputCollector` type can handle arbitrarily many key-value pairs.

4.1 Further Notes

- The Hadoop types `Text`, `LongWritable`, and `IntWritable` represent text and integer values in formats that can be communicated through Hadoop's framework stages. Also, the method `OutputCollector.collect()` adds a key-value pair to an OutputCollector instance like `output`.
- *Note on scalability:* Our `reduce()` method consolidates all the ligands with a given score into a single string (transmitted as `Text`), but this appending of strings does not scale to very large data. If, for example, trillions of ligand strings are possible, then `reduce()` must be revised. For example, one might use a trivial reducer that will produce a fresh key-value pair for each score and ligand, effectively copying key-value pairs to the same key-value pairs. Automatic sorting services provided by Hadoop between the "map" and "reduce" stages will ensure that the output produced by the "reduce" stage is sorted by the `key` argument for calls to `reduce()`. Since those `key` arguments are scores for ligands in our application, this automatic sorting by `key` makes it simpler to identify the ligands with large scores from key-value pairs produced by that trivial reducer.

4.2 Questions for exploration

- Try running the example `dd_hadoop.java` on a system with Hadoop installed.
  - This code does not generate data for the "map" stage, so you will have to produce your own randomly generated ligands, perhaps capturing the output from `Generate_tasks()` for one of the other implementations.
  - Once you have a data set, you must place it where your Hadoop application can find it. One ordinarily does this by uploading that data to the Hadoop Distributed File System (HDFS), which is typically tuned for handling very large data (e.g., unusually large block size and data stored on multiple disks for fault tolerance).
  - Rename the source file to `DDHadoop.java` (if necessary) before attempting to compile. After compiling the code, packaging it into a .jar file, and submitting that Hadoop job, you will probably notice that running

the Hadoop job takes far more time than any of our other implementations (including sequential), while producing the same results. This is because the I/O overhead used to launch a Hadoop job dominates the computation time for small-scale data. However, with data measured in terabytes of petabytes, it prepares for effective computations in reasonable time (see Amdahl's law).

  – Hadoop typically places the output from processing in a specified directory on the HDFS. By default, if the "map" stage generates relatively few key-value pairs, a single thread/process performs `reduce()` calls in the "reduce" stage, yielding a single output file (typically named `part-00000`).

- Modify `dd_hadoop.java` to use a trivial reducer instead of a reducer that concatenates ligand strings. Compare the output generated with a trivial reducer to the output generated by `dd_hadoop.java`.
- Research the configuration change(s) necessary in order to compute with multiple `reduce()` threads/processes at the "reduce" stage. Note that each such thread or process produces its own output file `part-NNNNN`. Examine those output files, and note that they are sorted by the `key` argument for `reduce()` within each output file.
- Would it be possible to scale one of our other implementations to compute with terabytes of data in a reasonable amount of time? Consider issues such as managing such large data, number of threads/nodes required for reasonable elapsed time, capacity of data structures, etc. Are some implementations more scalable than others?
- For further ideas, see exercises for other parallel implementations.

4.3 Readings about map-reduce frameworks and Hadoop

- [Dean and Ghemawat, 2004] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters, 2004.
- [Hadoop] Apache Software Foundation. Hadoop.
- [White, 2011] T. White, Hadoop: The definitive guide, O'Reilly, 2nd edition, 2011.

CHAPTER

FIVE

EVALUATING THE IMPLEMENTATIONS

5.1 Strategic simplifications of the problem

We consider the effects of some of the simplifying choices we have made.

- Our string-comparison algorithm for the "map" stage only vaguely suggests the chemistry computations of an actual docking algorithm. However, the computational complexity properties of our representative algorithm allow us to generate lengthy computation time by increasing the length of ligands (and having a long protein).
- Our implementations generate all of the candidate ligands before proceeding to process any of them. As mentioned in exercises, it might be reasonable to generate new ligands as a result of processing. The implementations `dd_serial.cpp` and `dd_boost.cpp` use a queue of ligands to generate the "map" stage work, and could be adapted to enable new ligands to be generated while others are being processed. We could also modify the Go implementation `dd_go.go` similarly, since we could dynamically add new ligands to the channel `ligands`.
- The amount of time it takes to process a ligand depends greatly on its length. This sometimes shows up in tests of performance: testing a *few* more ligands might require a *great deal* more time to compute. This may or may not fit with the computational pattern of a realistic docking algorithm. If one wants to model more consistent running time per ligand, the minimum length of ligands could be raised or lengths of ligands could be held constant.

5.2 The impact of scheduling threads

The way we schedule work for threads in our various parallel implmenentations may have a sizable impact on running time, since different ligands may vary greatly in computational time in our simplified model.

- By default, OpenMP's `omp parallel for`, as used by `dd_omp.cpp`, presumably divides the vector of ligands into roughly equal segments, one per thread. With small `nligands`, if one segment contains more lengthy ligands than another, it may disproportionately extend the running time of the entire program, with one thread taking considerably longer than the others. With large `nligands`, we expect less variability in the computational load for the threads.
- In our Boost thread implementation `dd_boost.cpp`, each thread draws a new ligand to process as soon as it finishes its current ligand. Likewise, the Go code `dd_go.go` draws ligands from the channel named ligands . This scheduling strategy should have a load-balancing effect, unless a thread draws a long ligand late in the "map" stage. One might try reordering the generated ligands in order to achieve better load balancing. For example, if ligands were sorted from longest to shortest before the "map" stage in the Boost thread implementation, the amount of imbalance of loads is limited by the shortness of the final ligands.

5.3 Barriers to performance improvement

The degree of parallelism in these implementations is theoretically limited by the implicit barrier after each stage of processing.

- In all of our implementations, the task generation stage produces all ligands before proceeding to any the "map" stage. In a different algorithm, parallel processing of ligands might begin as soon as those ligands appear in the task queue. We wouldn't expect much speedup from this optimization in our example, since generating a ligand requires little time, but generation of tasks might take much longer in other problems, and allowing threads to process those tasks sooner might increase performance in those cases.
- The "map" stage produces all key-value pairs before those pairs are sorted and reduced. This barrier occurs implicitly when finishing the `omp parallel for` pragma in our OpenMP implementation `dd_omp.cpp`, and as part of the map-reduce framework Hadoop used by `dd_hadoop.java`. That barrier appears explicitly in the loop of `join()` calls in our Boost threads code `dd_boost.cpp`. At the point of the barrier, some threads (or processes) have nothing to do while other threads complete their work.
- Perhaps threads that finish early could help carry out a parallel sort of the pairs, for better thread utilization, but identifying and implementing such a sort takes us beyond the scope of this example.
- The other stages are executed sequentially, so the "barrier" after each of those stages has no effect on computation time.

5.4 The convenience of a framework

Using a map-reduce framework such as Hadoop enables a programmer to reuse an effective infrastructure of parallel code, for greater productivity and reliability. A Hadoop implementation hides parallel algorithm complexities such as managing the granularity, load balancing, collective communication, and synchronization to maintain the thread-safe task queue, which are common to map-reduce problems and easily represented in a general map-reduce framework. Also, the fault-tolerance properties of Hadoop make it a scalable tool for computing with extremely large data on very large clusters.

5.5 Looking ahead: Parallel patterns

- Structural and computational patterns (Application architecture level): Map-reduce is a structural pattern. Our map-reduce algorithms represented in `MR::run()` methods for parallel implementations use a Master-worker structural pattern, in which one thread launches multiple worker threads and collects their results.
- **Program structure patterns:**
    - We use the Strict data parallelism pattern in parallel implementations of this exemplar, in which we apply our `Map()` algorithm to each element of the task queue (or vector) for independent computation.
- Implementation strategy patterns:
    - In the case of OpenMP and Hadoop, the master-worker computation is provided by the underlying runtime or framework. In addition, the Boost threads code exhibits an explicit Fork-join program-structure pattern. The OpenMP code's `omp parallel for` pragma implements the Loop parallel program-structure pattern, as does the Boost threads code with its `while` loop, and the Go implementation with its `for` loop in its "map" stage. In addition, Hadoop proceeds using an internal Bulk synchronous parallel (BSP) program-structure pattern, in which each stage completes its computation, communicates results, and waits for all threads to complete before the next stage begins. The `MR::run()` methods of our C++ parallel implementations for multicore computers also wait for each stage to complete before proceeding to the next,

which is similar to the classical BSP model for distributed computing. The Go implementation exhibits BSP at both ends of its sort stage, when it constructs an array of all pairs and completes its sorting algorithm. Most of our implementations use a Task queue program-structure pattern, in which the task queue helps with load balancing of variable-length tasks.

  – Besides these program-structure patterns, our examples also illustrate some *data-structure* patterns, namely Shared array (which we've implemented using TBB's thread-safe `concurrent_vector`) and Shared queue (TBB's `concurrent_bounded_queue`). Arguably, the use of channels `ligands` and `pairs` in the Go implementation constitutes a Shared queue as well.

- We named our array of threads `pool` in the Boost threads implementation in view of the Thread pool pattern for advancing the program counter. Note that OpenMP also manages a thread pool, and that most runtime implementations of OpenMP create all the threads they'll need at the outset of a program and reuse them as needed for parallel operations throughout that program. Go also manages its own pool of goroutines (threads). The Go example demonstrates the Message passing coordination pattern. We used no other explicit coordination patterns in our examples, although the TBB shared data structures internally employ (scalable) Mutual exclusion in order to avoid race conditions.

Note: Having developed solutions to our drug-design example using a pattern methodology, we emphasize that methodology does not prescribe one "right" order for considering patterns. For example, if one does not think of map-reduce as a familiar pattern, it could make sense to examine parallel algorithmic strategy patterns before proceeding to implementation strategy patterns. Indeed, an expert in applying patterns will possess well-honed skills in insightfully traversing the hierarchical web of patterns at various levels, in search of excellent solutions.