CSE211: Compiler Design

Homework 4: Parallel Schedules and DAE
Assigned: Dec. 3, 2020
Due: Dec. 14, 2020

- Read through instructions fully before beginning assignment; the testing scripts described later can help you while developing!
- These assignments are done with Python 3; you'll need to explicitly call `python3` on the server unless you switch it in your environment. The default `python` command defaults to version 2.7.
- The first two parts produce C++ files which are then compiled by `clang++`. Please note I have built a debug version of clang, so compiling is probably slower than you are used to. Please don't be alarmed.
- This homework requires timing experiments! Please only run experiments on one thread at a time. Before you start an experimental campaign, please run `top`. If more than 4 people are running experiments (look at how much memory and CPU are being used), then wait for an hour or so. If resources become a major constraint, we can set up a reservation system.

1 DOALL Loop Parallel Schedules

Here we will consider different ways to parallelize DOALL loops, also known as a *parallel schedule*. Static work partitioning works well for DOALL loops where iterations take roughly the same amount of time. When loop iterations have more variation, it helps to use dynamic scheduling, e.g. workstealing. Dynamic strategies are further parameterized by the granularity of tasks and if worklists are local or global. Once a loop is proven safe to parallelize, its the compilers job to pick a parallel schedule and implement the parallelism.

Your assignment is to generate a SPMD (single program, multiple data) function that implements the following loop (given inside a function to ease your job):

```
void function(float *result, int * power, int size) {
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < power[i] - 1; j++) {
      result[i] = result[i] * result[i];
    }
  }
}
```

Notice that the outermost loop is safe to parallelize because the inner loop only reads and writes to arrays at index `i`. Each outer loop `i` computes the `power[i]`-th power of `result[i]` (assuming a greater-than-zero values in `power`). Depending on the values in `power`, there is potentially load imbalance across loop iterations. There are functions to instantiate `power` with various work distributions and you will measure the speedups provided by different schedules.

1.1 Preliminaries

1. Review the slides from November 19th about implementing different parallel schedulers.
2. Although not required for the homework, you can review the available schedules in OpenMP. They are similar the schedulers you will be implementing: http://www.inf.ufsc.br/~bosco.sobral/ensino/ine5645/OpenMP_Dynamic_Scheduling.pdf
3. Similar to the previous homeworks, a skeleton and a utility header located here:

   `/home/tsorensen/public/homework4/part1`.

   Make a copy over to some homework4/part1 directory within your home directory. e.g. run the following:

   `cp -r /home/tsorensen/public/homework4/part1/* ./`
4. The coding aspect of this assignment is constrained largely to `skeleton.py`. Read through this code to understand the structure. The python code is writing a C++ file that is then compiled with `clang++` and executed. The C++ file will time (and validate) your implementation loop against a reference.
5. you should be able to run the skeleton as it is initially. You will fail the assertion check when running the C++ program, as the homework loop is not implemented. Examine `homework.cpp` to see the structure.

1.2 Technical work

You will write python code to fill in the loop body of:

```
void parallel_func(float *result, int * power, int size, int tid, int num_threads)
```

The `main` function will launch this function with 4 threads (the number of cores on our server), instantiating the `tid` and `num_threads` arguments. You can assume thread numbers are unique for threads and are contiguous, i.e. 0, 1, 2, 3. You can assume `size` is a power of 2.

The skeleton contains various schedulers that you are required to implement.

- `Static Scheduler`: Implement a scheduler that evenly divides outer iterations among threads. Use a chunking strategy, i.e. the first thread gets the first 1/4 of the iterations.
- `Global Workstealing Scheduler`: Implement a scheduler that dynamically assigns loop iterations to threads. You should do this keeping track of the index with a C++ `atomic_int` and atomic updates. If you are unfamiliar with C++ concurrency, please review here:

  https://medium.com/swlh/c-multithreading-and-concurrency-introduction-f640ce986fa7

This scheduler takes an additional integer argument specifying the granularity of how many iterations to take from the global iteration counter at a time. That is, a thread might update the global counter by N, and then perform N iterations locally. For example, a granularity size of 8 would atomically increment the global counter by 8. This will return a value, say 128. The thread would then perform iterations 128 through 136 locally, i.e. without performing any atomic updates. When these iterations are finished, the thread would atomically add 8 to the global counter again, followed by 8 local iterations. This reduces the contention on the global counter at the expense of less load balancing.

- `Local Workstealing Scheduler`: Implement a scheduler with local worklists. Given that we are parallelizing over contiguous ranges, we can simplify our worklists to be keep track of a contiguous range and location inside the range. I have implemented such a structure for you called `concurrent_range`, with the following API:

  - `void init(int start, int end)`: initialize the concurrent range with a start and end index. This is not thread safe. Do this before the threads are launched.
  - `bool dequeue(int &index, granularity)`: dequeue a range of values. The function returns a bool indicating if the queue was successful or not. You can assume any failed dequeue is because the range does not contain enough elements to satisfy the dequeue. The index dequeued is returned through the index argument (passed by reference). The granularity indicates how many values to dequeue at a time. This function is thread safe.

  Initialize each worklist with an equal number of elements. Once a thread finishes its work, allow it to try and steal work from another thread.

All schedulers can be implemented by inserting code in three places, for which I have provided functions (see skeleton code):

- global variables
- before the threads are launched (`pre_parallel_code`)
- the SPMD function body

These three functions can be found between lines 80 and 100. This is marked with BEGIN/END HOMEWORK comments.

1.3 Evaluation

The `generate_and_run` function additionally takes an arguement to specify the work distribution. There is

- uniform: all loop iterations take roughly the same amount of time, i.e. the compute the same power.
- linear: work increases linearly with `i`. That is, later values of `i` compute a higher power of `result[i]`.

- squared: work increases with `i` squared
- cubed: work increases with `i` cubed

each level provides an increasing level of load imbalance for your evaluation.

Evaluate each scheduler on each type of work distribution. For the dynamic schedulers, experiment with different granularities (please keep as a power of 2 to ease your implementations). Report on the trend of granularities, i.e. up to what value it provides a benefit with the different work distributions. This should include graphs and written descriptions.

Summarize your results. What are the pros and cons of the different approaches? When do they work best and why?

1.4 Submission

Please create a directory in your home directory called: `submissions`. Within that directory, create a directory called `homework4_part1`. Move your `skeleton.py` into that directory along with a PDF your graphs and observations.

2 Decoupled Access Execute (DAE) Slicing

In this part of the assignment, you will provide a (DAE) slicing of a basic block of code. We will use a slightly upgraded version of the instruction and basic block classes that we used in the local value numbering part of homework 2. The numbering scheme will be used to create SSA code. You will then insert DAE API calls and slice the program accordingly. This assignment considers onla a single basic block, so there is no need to consider control dependencies; only data dependencies.

- Review the slides from Dec. 1 about implementing DAE slicing.
- If interested, review the original DAE (1982) paper:
  `https://courses.cs.washington.edu/courses/cse590g/04sp/Smith-1982-Decoupled-Access-Execute-Computer-Architectures.pdf`
- If interested, review the DeSC (2015) paper:
  `https://mrmgroup.cs.princeton.edu/papers/taejun_micro15.pdf`
- Similar to the previous homeworks, a skeleton is located here:
  `/home/tsorensen/public/homework4/part2`.

  Make a copy over to some homework4/part1 directory within your home directory. e.g. run the following:

  ```
  cp -r /home/tsorensen/public/homework4/part2/* ./
  ```
- The coding aspect of this assignment is constrained to `skeleton.py`. Read through this code to understand the structure. This is similar to the BasicBlock and Instruction code for the local value numbering in Homeowork 2. I have extended the instructions to include memory instructions (loads and stores), as well as the DAE API calls: SB store value/addr, and FIFO enqueue/dequeue.
- you should be able to run the skeleton as it is initially, but you will fail the assertions. There are some simple tests cases in the skeleton file.

2.1 Technical Work

Read through `skeleton.py` carefully. The top-level function `decouple_block` takes a basic block and returns three values: an Access basic block, an Execute basic block, and the number of loads in the Access block that could be upgraded to *terminal loads*. I have outlined the various functions for you to implement:

- Replace memory instructions in the Execute with `SB_store_value` and `FIFO_dequeue` instructions.
- Add `FIFO_enqueues` instructions to the Access and replace stores with `SB_store_addr` instructions.
- Instantiate the slicing criteria for the Access and the Execute.
- Implement a general slicing function that takes a basic block and slicing criteria.
- Analyze the Access basic block and determine how many loads could be promoted to *terminal loads*.

2.2 Evaluation

Similar to prior works, I have provided a tester script that runs your program against a suite of automatically generated basic blocks. It will check the length of your access and execute basic blocks, and the number of loads that can be identified as terminal. Please make sure your code passes the tester and the console output is easily parsed (i.e. please limit any additional print statements and use types that pass the tester).

2.3 Submission

Please create a directory in your home directory called: `submissions`. Within that directory, create a directory called `homework4_part2`. Move your `skeleton.py` into that directory. Please make a hard copy (no symbolic links).