CS 33, Winter 2018
Parallel Performance Lab
Assigned: Feb 22nd
Due: March 14th, 11:55PM

1 Introduction

This assignment deals with applying performance and parallelism optimizations to data-intensive code. In this lab we will consider the problem of sorting key-value pairs, where keys and values are random unsigned integers.

Specifically, we will be implementing the the `RadixSort` kernel. Why RadixSort? 1. It's fast. (as you'll see, it's faster than the GNU standard library implementation of std::sort) 2. It's fairly parallelizable. 3. It's very CS33-y – it uses bit manipulation to find extract bits for the counting and distribution phases. For more general information on the RadixSort kernel, you can see the Wikipedia page:

https://en.wikipedia.org/wiki/Radix_sort

We will be using the LSD (least significant digit first) version of Radix sort in this lab. Roughly it works in three phases.

- **Histogram Phase:** Scan a subset of the bits (length of the radix) of all elements in the list, and count the number of occurrences of each digit – place these in a histogram.
- **Prefix-sum Phase:** Perform a prefix-sum on the histogram to find out where each digit should go inside the overall list.
- **Data-distribution Phase:** Distribute each key-value pair to the position indicated by the prefix-sum array, and update this array so the next occurrence of that digit goes in the subsequent location.

This is performed repeatedly, each time looking at more significant digits, until all digits have been examined, and the final sorted result is computed.

2 Lab Setup

Please work individually, you will only hand in the kernels.c file to CCLE, similar to lab 1.

The lab will be here: /w/class.1/cs/cs33/csbin

Start by copying `perflab-handout.tar` to a directory in which you plan to do your work. Then give the command: `tar xvf perflab-handout.tar`. This will cause a number of files to be unpacked into the directory. The only file you will be modifying and handing in is `kernels.c`. The `driver.c` program is a driver program that allows you to evaluate the performance of your solutions. Use the command `make driver` to generate the driver code and run it with the command `./driver`.

Looking at the file `kernels.c` you'll notice a C structure `user` into which you should insert the requested identifying information. **Do this right away so you don't forget.**.

3 Implementation Overview

Data Structures

The core data structure is for representing key value pairs (kvp's). A `kvp` is a struct as shown below:

```
typedef struct {
   unsigned int red;   /* key to sort on */
   unsigned int green; /* associated value */
} kvp;
```

As can be seen, keys and values have 32-bit representations. An array to be sorted is represented as a one-dimensional array of `kvp`. See the file `defs.h` for this code.

Radix Sort

There are two implementations for you to complete, a single-threaded and multi-threaded version of radix sort. Both functions have the same interface.

They take as input a source kvp array `src` and return the result in the destination array `dst`. The function is free to overwrite the source array, which is helpful since radix-sort is commonly implemented as an out-of-place algorithm. The parameter `dim` indicates the length of both of these arrays. (do not go past the dimensions of these arrays, or your program will likely error).

Here is the signature of the naive (low-performance) implementation that you will start with:

```
void naive_singlethread(int dim, kvp *src, kvp *dst) {
  ...
}
```

The naive version of the code uses the number 256 (2^8) as the radix. This was empirically found to balance histogram and prefix-sum phases of the computation.

Performance measures

Our main performance measure is *CPE* or *Cycles per Element*. If it takes C cycles to run for an array size of N, the CPE value is C/N. (Note that radix sort is O(N), because it is not a comparison based sorting algorithm).

The ratios (speedups) of the optimized implementation over the naive one will constitute a *score* of your implementation. To summarize the overall effect over different values of N, we will compute the *geometric mean* of the results for 7 array sizes: 1024 4096 16384 65536 262144 1048576 4194304.

Input Assumptions

To make life easier, you can assume that N is a multiple of 32. Your code must run correctly for all such values of N.

Baseline

The baseline implementation is the GNU standard c++ library's std::stable_sort, which probably uses a merge sort. There is some code commented out in the naive version if you want to try it out.

We will measure your performance on lnxsrv07, as it has a pretty powerful processor in it (although it's a bit old). Both lnxsrv07 and lnxsrv09 have this processor, so testing on one of these machines is recommended. The baseline CPE values are measured and fixed, so the speedup will not be very meaningful on any other machine.

If you want to find the core count/type of the machine, run *cat /proc/cpuinfo*.

4 Infrastructure

We have provided support code to help you test the correctness of your implementations and measure their performance. This section describes how to use this infrastructure. The exact details of each part of the assignment is described in the following section.

Note: The only source file you will be modifying is `kernels.c`. Some helper code is in *helper.c*.

Versioning

You will likely be writing many versions of the code. To help you compare the performance of all the different versions you've written, we provide a way of "registering" functions.

For example, the file `kernels.c` that we have provided you contains the following function:

```
void register_singlethread_functions() {
   add_singlethread_function(&singlethread, singlethread_descr);
}
```

This function contains one or more calls to `add_singlethread_function`. In the above example, `add_singlethread_function` registers the function `singlethread` along with a string `singlethread_descr` which is an ASCII description of what the function does. See the file `kernels.c` to see how to create the string descriptions. This string can be at most 256 characters long.

A similar function for your singlethread kernels is provided in the file `kernels.c`.

Driver

The source code you will write will be linked with object code that we supply into a `driver` binary. To create this binary, you will need to execute the command

```
unix> make driver
```

You will need to re-make driver each time you change the code in `kernels.c`. To test your implementations, you can then run the command:

```
unix> ./driver
```

The `driver` can be run in four different modes:

- *Default mode*, in which all versions of your implementation are run.
- *Autograder mode*, in which only the final version is run. This is the mode we will run in when we use the driver to grade your handin.
- *File mode*, in which only versions that are mentioned in an input file are run.
- *Dump mode*, in which a one-line description of each version is dumped to a text file. You can then edit this text file to keep only those versions that you'd like to test using the *file mode*. You can specify whether to quit after dumping the file or if your implementations are to be run.

If run without any arguments, `driver` will run all of your versions (*default mode*). Other modes and options can be specified by command-line arguments to `driver`, as listed below:

`-g` : Run only the final `singlethread()` and `multithread()` functions (*autograder mode*).

`-f <funcfile>` : Execute only those versions specified in `<funcfile>` (*file mode*).

`-d <dumpfile>` : Dump the names of all versions to a dump file called `<dumpfile>`, *one line* to a version (*dump mode*).

`-q` : Quit after dumping version names to a dump file. To be used in tandem with `-d`. For example, to quit immediately after printing the dump file, type `./driver -qd dumpfile`.

`-h` : Print the command line usage.

Identity Information

Important: Before you start, you should fill in the struct in `kernels.c` with information about yourself. This information is just like the one for the Data Lab.

5 Assignment Details

Grading (out of 80 points)

In this part, you will optimize the `singlethread` and `multithread` version to achieve as low a CPE as possible for both versions. The grade will be calculated as:

- 20 Points for singlethread: min((Speedup-1) / 4.0, 4.0) * 20
- 60 Points for multithread: min((Speedup-1) / 8.0, 8.0) * 60

In otherwords, your target speedup is 4x for `singlethread`, and 8x for `multithread`, and your grade is simply the speedup divided by that target.

There are 3 opportunities for bonus points:

- Top 25% of students with best overall multithreaded speedup will get 10 bonus points.
- Top 10 singlethread performance will get additional 5 points bonus.
- Top 10 multithread performance will get additional 5 points bonus.

The top 10 solutions will obviously be given greater scrutiny in terms of following the rules. :)

Some Advice

For the traditional singlethread optimizations, remember to focus on optimizing the inner loops using the optimization tricks covered in class. That means loop unrolling, function inlining, cache locality (eg. through loop-reordering), etc. You may even want to look at the assembly code generated, though it is not required.

To multi-thread the code, we recommend using pthreads. The pthread libraries are already included, so you should be able to use them without modifying the makefiles or code. Remember, you'll need to partition the work across different threads – there are lots of choices for how to do this, some are more cache friendly than others, and some introduce extra dependences and race conditions!

The basic outline is:

- Create several threads, store their ids in an array for later.
- Pass each thread an "id" indicating which part of the sorted kvp array to work on.
- Do iteratively in parallel:
  - Do histogram in parallel (thread barrier)
  - Do prefix sum in parallel (thread barrier)
  - Do do data distribution in parallel (thread barrier)
- Join the threads using the saved ids to wait until they are complete.

Keep in mind that the optimal number of threads for any given matrix size might be different. If your matrix is small, the overheads of starting many threads might actually completely outweigh the benefits of multithreading!

Coding Rules

You may write any code you want, as long as it satisfies the following:

- It must be in ANSI C. You may not use any embedded assembly language statements. Note that though the makefile uses the g++ compiler, which is a C++ compiler, we only do that so that we can compare against std::stable_sort. Your implementation should be in C.
- It must not interfere with the time measurement mechanism. You will also be penalized if your code prints any extraneous information.
- You must NOT DEADLOCK. You will get 0 points if you deadlock. (if your code never finishes)

- One of the multi-threaded invocations must actually use multi-threading.

You can only modify code in `kernels.c`. You are allowed to define macros, additional global variables, and other procedures in these files.

Academic Honesty

If you work with anyone in this project, please indicate that in the provided space in kernels.c.

Of course, your entire file should be hand-written by you. You are free to look at tutorials and academic literature for radix-sort based sorting. You are *not* allowed to look at or copy from implementations on online code repositories like github. (!)

6 Hand In Instructions

When you have completed the lab, you will hand in one file to CCLE, `kernels.c`, that contains your solution.

- Make sure you have included your identifying information in the struct in `kernels.c`.
- Make sure that the `singlethread()` function and `multithread()` function corresponds to your fastest implementations – we will grade these two only.
- Remove any extraneous print statements.
- If it does not compile and run on lnxsrv07, you will get a zero. :)

Good luck!