CS410 Assignment 1: Parallel Merge Sort Using Cilk

Date released: Feb/1/2024.
Due date: Feb/15/2024 (11:59 pm IST).

In this assignment, you will implement a **parallel merge sort using Open Cilk.**
You will use the **DGX cluster** to compile and run your code.
You may do the project in a group of two students.
You may discuss ideas but do not share the code with another team.

Distribution and collection of assignments will be managed using **GitHub Classroom**. You can accept the assignment at the following URL: https://classroom.github.com/a/gzquYHNt

When you accept the assignment, it will provide you with a clone of a GitHub repository that contains a copy of a sequential program that implements a serial merge sort on 64-bit integers. You may use this program as you see fit to get a jump start on your assignment. You may include any or all of the provided code in your submitted program. Feel free to use the code directly as the basis for your parallel solution.

Write a shared-memory parallel program in Open Cilk that sorts 64-bit integers using merge sort. The program should accept two arguments:

- first, the number of elements to sort,
- and second a cutoff value for parallelism.

The merge sort uses O(N) extra memory, which is included in the signature of `MergeSort` function.

A brief description of the provide code (sort.serial.cpp):

1. `void MergeSort(vector<int64_t> & vec, int64_t start, vector<int64_t> & tmp, int64_t tmpStart, int64_t sz, int64_t depth, int64_t limit)` sorts `vec[start..start+sz-1]` in the increasing order. It uses `tmp[tmpStart…tmpStart+sz-1]` as a temporary array. The `depth` and `limit` arguments are unused in this serial version but you will use them in your parallel version. The serial version cuts the array into two halves and sorts them separately. **You MUST write a parallel version, which will do something more as described in the class.** The two sorted halves are merged using `SerialMerge()` into the `tmp[]` vector. The resulting sorted `tmp` is copied back into `vec`.
2. `SerialMerge(vector<int64_t> & vec, int64_t Astart, int64_t Aend, int64_t Bstart, int64_t Bend, vector<int64_t> & tmp, int64_t tmpIdx)` function merges two sorted arrays `vec[Astart..Aend]` and

`vec[Bstart..Bend]` into vector `tmp` starting at index `tmpIdx`. It follows a simple algorithm that uses two cursors one on each sorted vectors and copies an element that is smaller between the two to `tmp` and advances the corresponding cursor. **This serial version MUST be parallelized as described in the class.**

3. `Validate(const vector<int64_t> & input)` function is a simple validation that ensures that every ith element is greater than or equal to (i-1)th element of `input`. It counts the number of times this criterion is violated. **You MUST parallelize this part.**
4. The `Init(vector<int64_t> & vec, int64_t start, int64_t end)` function initializes `vec[start..end-1]` with random 64-bit integers. **You MUST parallelize this also.**

The main function takes the vector size and a cutoff value, initializes the vector, calls `MergeSort()`, and validates the result. The program also measures the running time of `MergeSort()` in milliseconds. This is the most critical part of the assignment – parallelize this first.

Experiments

Basics

Your submitted program should be able to sort **2 billion 64-bit integers** in a reasonable time (less than a minute with 32 cilk workers).

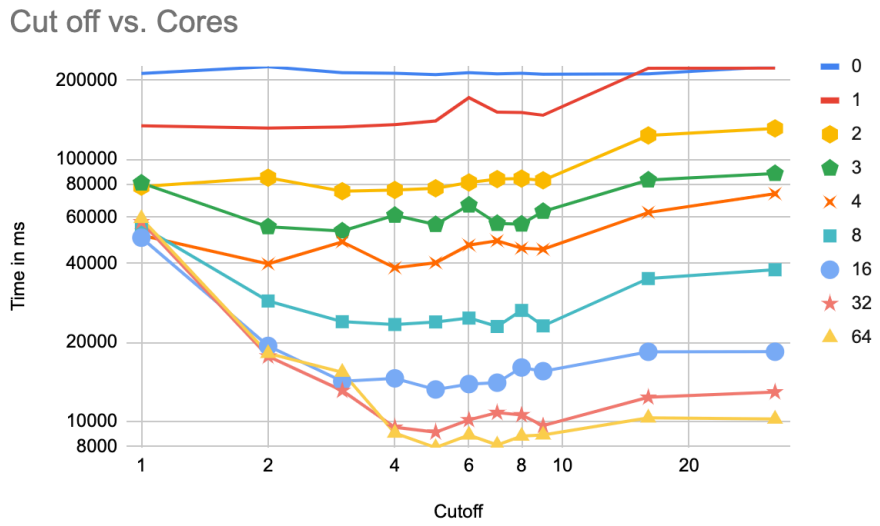
Data race detection

Your submitted program should be free of data races. Open Cilk's `cilksSan` uses code rewriting to instrument your executable to check itself for data races as it runs. Running your instrumented program at the front of the command line will check an execution for data races. If cilkSan reports races, make sure that you compile your program with the -g flag and run it again. (Executables compiled with -g have more detailed race reports, which will help you identify the references involved in the data races.) The provided Makefile (`sortSan` target) shows that you need to pass "`-fsanitize=cilk`" flag to the compiler to instrument the code. The data race detection should be run with **CILK_NWORKERS=1.** See the `runSortSan` target in the Makefile.

Running time

Compile your program using -O3 optimization. All of the execution measurements described below should be performed with this version of your executable.

Run the program with **2 billion input size**. Vary CILK_NWORKERS over 1, 2, 4, 8, 16, and 32. Vary the cut off over 0, 1, 2, 3, 4, 5, 6, 7, 8, 16, and 32 and plot the `MergeSort` time in milliseconds.
Prepare a graph of the running time of `MergeSort` in milliseconds. Plot a point for each of the executions. The X axis should show the cutoff; the Y axis should show the running time in ms. Add one line for each of the CILK_NWORKERS values as shown below.



CilkScale

Open Cilk's cilkScale tool instruments your program to profile its parallelism. cilkScale will report the *span bound*, *perfect linear speedup*, *observed speedup*, and *burdened dag bound*. Use cilkScale to profile your program and record your measurements. Include the measurement results from cilkScale in your report in a table. Ideally, your observed numbers should be between the *burdened dag bound* and *span bound.* Use this tool/graph to tune your code.

To run cilkScale you need to produce two version of your code one with "`-fcilktool=cilkscale`" flag (see the `sortScale` target in the Makefile) and another with "`-fcilktool=cilkscale-benchmark`" flag (see the `sortBenchmark` target in the `Makefile`). To run the tool, you need to invoke the `cilkscale.py` script. See the `runSortScale` target in the Makefile. Here is an example launch of cilkScale:

```
python3
/home/iit/cs410software/opencilk_2_0_0/share/Cilkscale_vis/cilkscale.
py -c ./sortScale -b ./sortBenchmark -cpus 1,2,3,4,8,16,32
--output-csv report.csv --output-plot report.pdf -a 1000000000 10
```

It produces a CSV report (report.csv) and a PDF plot (report.pdf). Include the plot in your report. Produce a graph containing information from cilkScale for **1 billion size vector** with a few different cutoff values in your report.

Parallel efficiency

Prepare a graph of the efficiency of your parallelization comparing the real times measured on 1-32 cores. Plot a point for each of the executions. The X axis should show the number of cores. The Y axis should show your measured parallel efficiency for the execution. **Pick the cutoff that works best for you.** Parallel efficiency is computed as S/(p * T(p)), where S represents the real time of a sequential execution of your program, p is the number of processors and T(p) is the real time of the execution on p processors. Discuss your efficiency findings and the quality of the parallelization.

Note: Don't make the mistake of using the execution time of a one thread version of your parallel implementation as the time of a sequential execution. The execution time of the one-thread parallel implementation (T_1) is typically larger than that of the original sequential (T_s) implementation. When you compute parallel efficiency, always use the performance of a sequential code as a baseline (T_s); otherwise, you will overestimate the value of your parallelization.

The sequential execution time of your Open Cilk program is not the same as the execution time of an Open Cilk program with one worker. To provide a baseline for your parallel efficiency measurements, you can create a serialization of your program that will execute sequentially without the overhead of the Open Cilk runtime by **NOT** passing the `-DCILK` flag to the compiler. When you compile your serialized code, make sure to use -O3 optimization. The `sortSerial` target in the Makefile produces the serial version by eliding the cilk keywords (you may have to adjust a few more things depending on your code).

Written report

Write a report that describes how your program works. This report should not include your program, though it may include one or more figures containing pseudo-code that sketches key elements of the parallelization strategy. Explain how your program exploits parallelism. Explain why you chose to exploit parallelism in this way. Were there any opportunities for parallelism that you chose not to exploit? If so, why? Explain how the parallel work is synchronized. Discuss and explain the differences you observe in the profiles with different cutoff depth. Include all results for the sections described in the Experiments section above.

Submitting your assignment

Your assignment should be submitted in two parts.

- One or more source files containing your Open Cilk program. Your submission must contain a Makefile with the following targets:
    - `sortSerial`: builds the serial version of code with NO cilk usage.
    - `runSortSerial`: runs your serial code with the necessary arguments.
        - For example, `make runSortSerial SIZE=100000 CUTOFF=10` should run the serial sort on 100K integers with a cutoff of 10.
    - `sortCilk`: builds the cilk version of your code.
    - `runSortCilk`: runs your cilk code with the necessary arguments and environment variables.
        - For example, `make runSortSilk CILK_NWORKERS=4 SIZE=100000 CUTOFF=10` should run the serial sort on 100K integers with a cutoff of 10 while using 4 cilk workers.
    - `sortSan`: builds your cilk code instrumented to detect data races.
    - `runSortSan`: runs your cilk code with CILK_NWORKERS=1 and reports data races.
    - `sortScale`: compiles instrumented version for cilkScale.
    - `sortBenchmark`: compilers benchmark program for cilkScale.
    - `runSortScale`: runs cilkscale.py
- A report about your program in PDF format. Guidelines for your report: 3 pages won't have enough detail; more than 10 pages will say too much. Plan for a length in between. Make sure that you address all of the requirements specified in the "Experiments" section.

You will submit your assignment by committing your source files, Makefile, and report to your GitHub Classroom repository. If you have trouble using GitHub and come up against the deadline for submitting your assignment, email the instructor a tar file containing your source code and your report.

Click on the link shared above. This will create a repository in your GitHub account.

1. Clone the repository into your local development environment (on PNK Server) using the **git clone** command.
2. Add all assignment related files that you want to submit to your GitHub local repository using the **git add** command.
3. Save the changes using the **git commit** command
4. Upload the changes to GitHub using the **git push** command
5. Release your changes by first tagging your commit on the local environment using the **git tag -a cs410assignment1 -m “submitting assignment 1”**.
6. Next, push the tag to GitHub with the help of the following command: **git push --tags** \\

If you want to make changes after you have submitted (repeat the above steps from 1 to 5 and apply modified commands shown below in place of step 4):

git tag -a -f git tag -a cs410assignment1 -m ”submitting assignment 1”
git push -f --tags

Grading criteria

- 20% Program correctness. Program must correctly implement the merge sort as described in the class lecture. The submission must include all aspects of the Makefile (serial, parallel, datarace, and scalability evaluation targets).
- 10% No data races. Use CilkSan to prove the absence of races.
- 30% Program clarity, elegance and parallelization. The program should be well-documented internally with comments. Your program should have no data races.
- 10% Program scalability and performance.
- 30% Report. Your grade on the report will consider the quality of the writing (grammar, sentence structure), whether all of the required elements are included, and the clarity of your explanations. Make sure that you have performed all of the experiments requested above and provided the information requested about them in your report. If you haven't, you will lose points!

Using the DGX Server

1. `ssh c<your id>@10.250.101.55` through the terminal (e.g. if your id is `200010011` then you would execute the command `ssh c200010011@10.250.101.55`).
2. Log in with your id and password.

Using SLURM on DGX to submit jobs

While you can develop your program on one of the login nodes on DGX, you MUST run your experiments on one of the compute nodes using the SLURM job manager. You can obtain a private node to run your job on by requesting a node for an interactive session using

```
srun --pty --export=ALL --nodes=1 --ntasks-per-node=1
--cpus-per-task=32 --time=00:30:00 bash
```

A copy of this command is included in the file interactive among the provided files in your GitHub Classroom repository.

I strongly recommend developing and testing your code (with small vector size) on the login node to avoid the surprise of having your editor be killed as your interactive session on a compute node expires.

Your sample repository includes a SLURM batch script submit.sbatch that you can launch with SLURM's sbatch utility. The sample batch script runs the sort program. You can edit the file to run your code just once or run it multiple times in a sequence of experiments.
Example slurm script (run: `sbatch submit.sbatch`)

```
#!/bin/bash
#SBATCH --job-name=CILK_JOB          # Job name
#SBATCH --mail-type=END,FAIL          # Mail events (NONE,
BEGIN, END, FAIL, ALL)
#SBATCH --mail-user=email@iitdh.ac.in # Where to send mail
#SBATCH --nodes=1                     # Run on a single node
#SBATCH --ntasks-per-node=1           # Single task
#SBATCH --cpus-per-task=32            # 32 CPUs per task
#SBATCH --time=00:10:00               # Time limit hrs:min:sec
#SBATCH --output=serial_test_%j.log   # Standard output and
error log
pwd; hostname; date

set -ex
make all
make runSortCilk CILK_NWORKERS=32 SIZE=1000000000 CUTOFF=10
date
```

The default queue for the DGX machine is "v100". If we don't specify any queue name, v100 will be chosen. If v100 becomes overloaded, there is another queue "a100", which you can use. To explicitly specify the queue name, add the following extra line to batch script you submit to the sbatch command:

```
#SBATCH --partition=a100
```

General note:
If you see this message when using “-fcilktool=cilkscale-benchmark” or “-fcilktool=cilkscale” flags to compile, you can ignore it. It is just a general dump of a compilation step.

```
/home/iit/cs410software/opencilk_2_0_0/bin/clang++ -O3 -g
-std=c++11 -fcilktool=cilkscale-benchmark -fopencilk -DCILK -o
sortBenchmark sort.parallel.cpp
```

```
Sync   sync within %syncreg.i, label %sync.continue, !dbg
!2641has unwind
; Function Attrs: norecurse uwtable
define dso_local noundef i32 @main(i32 noundef %argc, i8**
nocapture noundef readonly %argv) local_unnamed_addr #17
personality i8* bitcast (i32 (...)* @__gxx_personality_v0 to
i8*) !dbg !3000 {
entry:
  %buffer.i.epil = alloca %struct.drand48_data, align 8
  %count.i = alloca i64, align 8
  %syncreg.i = tail call token @llvm.syncregion.start()
  %input = alloca %"class.std::vector", align 8
  %tmp = alloca %"class.std::vector", align 8
```