Term Project

Timeline

Email to 18645@sv.cmu.edu by 22:00(PST)

- Abstract: March 31th (1 page)
  - what you propose to do for this project
  - Baseline performance, expected outcome
  - submit early!
- Draft Project Report: April 16th (5-10 pages)
  - Project overview including preliminary results
- Final Project Report: April 23th (5-10 pages)
  - Final graded report: 5-10 pages

Report

Apply skills learned in this course to a problem that is important to you

- Technical report describing your work (5-10 pages)
  - 0.5 page: Overview of the problem you tackled. Why this problem?
  - 0.5 - 1 page: Literature survey, related work. What have others done?
  - 1 - 2 pages: Detailed description of techniques. What did you do?
  - 1.5 - 3 pages: Analysis of results What are the results? What do they mean?
  - 0.5 - 1 page: Conclusion. What is the take home message?
- Size of report should reflect size of team
- Report could be basis for a conference submission

Class

- location_tool: Read data
- location: Class
- pso_algo: algorithm
- reference_point: tool

Detailed description of techniques. What did you do?

Besides the method using CUDA, it is also possible to optimize the performance using OpenMP. CUDA can make better usage of the manycore computing power and OpenMP can boost the performance on multicore. Using both techniques can make the computation and test procedure faster in different ways. As we have many test cases, OpenMP can test them concurrently and CUDA can make each test case faster.

In this part we'll focus on the usage of OpenMP.

Though OpenMP may not be as efficient as the CUDA version on each test case, it can still boost the performance with little modification of the code as well as the data structure. This is also the advantage of OpenMP. It may have some strict limitation, but it can still be regarded to be a powerful and easy-to-use technique.

The development of a new algorithm consists of two parts: computing and testing. Not only should we put our concentration on computing, testing is also important. The procedure of our pso algorithm has these two parts, so what we want is to make them both faster.

However, simply adding several openmp statement can not meet our needs as most of the time it can not make full use of the powerful cores of our CPUs.

First we redesign our variables used in the algorithm so that less shared spaces are used as well as the sync issue. This is one of the most common techniques in the practice of OpenMP. For those statement that visits one of the elements of an array, we use a temporary variable to eliminate the false sharing. Also, the reduction operation is very useful.

```
double sum = 0, norm_a = 0, norm_b = 0;
#pragma omp parallel for reduction(+:sum, norm_a, norm_b)
for (int i = 0; i < (int)a.size(); i++) {
    sum += a[i] * b[i];
    norm_a += a[i] * a[i];
    norm_b += b[i] * b[i];
}
```

Loop unrolling is also one of the useful techniques that can reduce the amounts of instructions passed to the CPU. For those loops with limited circulating times, we replace the for statement with the complete statements. These methods increases the length of code but reduces the working load for CPU.

```
//origin
for(....){ statement section A } // 4 times
```

```
//after unrolling
statement section A;
statement section A;
statement section A;
statement section A;
```

However, we can't use OpenMP on all for statements as in the pso algorithm, the n+1 step needs the result of the n step, we have to add barriers to make sure that the computations are performed in the correct order.

So what we can do is to change our data structure so as to keep the correct order and reduce unnecessary operations. For pso algorithm, KD-tree is a good choice. However the implementation of KD-tree is difficult to suit the OpenMP. After tesing we have to give up combing KD-tree with OpenMP(But we use is on the CUDA version).

OpenMP is focusing on multi-platform shared memory multiprocessing programming. Using fork-join model, some STL containers don't work well in multithreading environment. That is to say, they are not thread-safe. So we modify our code in order to use the basic data structure such as array and struct. Actually, the random-access container(vector) works fine with OpenMP. But the set/map container is difficult to use correctly in OpenMP, even with the help of the latest version of OpenMP, only limited operations on these contains are available. We make most of the data structure friendly to OpenMP to accelerate the computation.

Analysis of results What are the results? What do they mean?

We test our code both on local machine and cmu ghc machine. Unlike CUDA, the number of cores really matters in OpenMP. Let's see the results first.

Method	Time(second)	Error	Speedup
Origin	216.7	1.71381	x
Local Machine(OpenMP)	51.365	2.22721	4.22x
GHC Machine(OpenMP)	11.942	1.74022	18.15x

Conclusion. What is the take home message?

- We have to make a choice between time and space, flexibility and speed
- Finding the opportunities of concurrency is the most difficult task in writing fast code

- The improvement on data structure may be better than any other optimizing methods
- However, sometimes the manycore/multicore optimization can be used on complex but efficient algorithm, so we have to choose a better way according to the situation.
-