Acknowledgement

I would like to express my deepest appreciation to all those who provided me the possibility to complete this report. A special gratitude I give to our project guide, Prof Leslie Lander whose contribution in stimulating suggestions and encouragement, helped me to work on this project and to successfully move further towards its completion.

Along with professor Lander. I am extremely grateful to for providing guidance for this project though he had busy schedule managing organizational affairs. I am also thankful to Mr. Dave Hall for providing me server access to allow me to complete the project

Signed: Date:

Synopsis

With the advent of multicore, multithreaded and high performing CPU cores in recent times, it has become imperative to see how they stack against each other and what it means for end user.

From an end user standpoint, having a faster core would mean getting work done quicker (provided that the drivers and resources allocated are same). For enterprise, it would mean less machine hours can be used thereby greatly reducing costs associated with running a server. With 2 primary operators in market for consumer and enterprise grade compute cores for PC's, namely Intel and AMD, there are various comparisons made regarding their relative performance. In domain of ARM we have Apples A series chipsets, Qualcomm's snapdragon chipsets as well as NVDIA's tegra chips. My focus would be on x86 from hereon while referring to chipsets. To do such comparisons, we use benchmarking tools. The goal of my project is to create one such benchmark tool "k-bench" that can be used to compare relative performance of 2 CPU's in an unbiased way. As another use case for my benchmark, I have also demonstrated effects of implementation of improper multi-threading and its consequences.

With AI and Machine Learning being at forefront of current technological advances, It becomes imperative to test out how would a system perform subjected to Machine learning tasks. One of basic clustering algorithms namely K-Nearest-Neighbors has been used to measure our system in regards to that metric.

Introduction

The basic purpose of benchmark is to measure the performance of a computer or device CPU (or SoC). A set of standards, or baseline measurements are used to compare the performance of different systems, using the same methods and circumstances [1].

To compare relative CPU performance, we need a baseline. For that purpose, I have used alpha.cs.binghamton.edu as a reference node. For benchmark to bring out optimal performance in CPU cores which are usually not meant to do graphical rendering or texture shading, I have used 2 different API's (C++11 and OpenMp) in completely different ways for the same purpose. I intend to make this benchmark as open source executable for those who want to test out performance of their PC's

The specifications of that PC cluster allocated to me are listed below

Specification Table	
Architecture	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order	Little Endian
Address sizes	36 bits physical, 48 bits virtual
CPU(s)	4
On-line CPU	0-3
Thread(s) per core	1
Core(s) per socket	2
Socket	2
Vendor ID	GenuineIntel
Model name	Intel(R) Xeon(R) CPU 5150 @ 2.66GHz

Implementation

For the project, as phase 1 I decided to do simple integer operations. For this purpose I inserted 500 million random integers to a std::vector in C++ and decided to add them to a single long double int. It is an example of embarrassingly parallel procedure (wherein CPU can easily parallelize requisite operations). So, I decided to test it out on 1 core and 4 cores using OpenMp and C++11 thread API respectively.

For purpose of demonstrating effects of incorrect use of threading, I made sure the values in OpenMP operated on same value repeatedly, bringing deadlocks in the fore and utilizing suboptimal amount of resources. I timed each operation and calculated MIPS (Million Instructions/Second) a standard measure of CPU operations.

To demonstrate C++11 threading API to show advantages of multithreading, I used std::reference to make sure that the values in multiple threads are operated by reference.

Since CPU's tend to drop or skip values while operating with large scale data on multiple cores, I compared results of all three summation outcomes and gave score to accuracy of each way of threading with 1 being perfectly symmetrical with sequential calculation performed beforehand.

Implementation II

Continuing from where I stopped in previous semester, I made a GUI for my application using c# and Visual basic. In meantime I tried my application on other PCs to get better idea of frame of reference I used in alpha.cs.binghamton server. Unfortunately, I lost connection to the server, and doing testing on my PC, I realized that testing on PC was different from testing on server. So, a web page with made with HTML and Javascript was ideal for me to make sure that my benchmark would reach as many people as possible. So, I moved to designing a simple web page in middle of my project. The page was as barebones as it was needed to be to avoid cluttering. The figure below represents entirety of the web page.

With On-Click button, a window pops up requesting user to run the benchmark (which is a downloadable file). On running it, you can see various parameters on which the PC was benchmarked against. I added K-NN algorithm as a code but due to server being down, I was unable to test it. I used a standard K-D tree to implement K-NN algorithm. I had done a similar benchmark in class of High-performance computing, but this time, instead of using the binary files as input/output, I decided to randomize point generation. Since this was untested on server. I have it in comments of my main code.

I have used a sample.exe to represent how my benchmark is expected to run. While end results might not be 100 percentage similar, they are expected to be an average case representation of my work.

Results

As we can see, there is a stark contrast between speeds of execution in all three methods. The multithreading implemented properly is up to 4 times faster than sequential operation, which incidentally is in linear proportion to number of cores/threads used by the program while OpenMp is 6 times slower than performance of single core.

We can also see that the accuracy is 1(or 100 percent) across the board which is imperative while doing any high value transaction

While I have not added K-NN here, since it is un-tested as of now, but, we can expect to see similar performance gains in execution of that benchmark too.

Performance Quantifiers	
Sequential Execution	Results
Sum,	249988686177578
Elapsed time	2.63517s
MIPS	190
OpenMP (Improperly Implemented)	Results
Sum	249988686177578
Elapsed time	21.857s
MIPS	23
Accuracy	100
C++11 Thread API	Results
Sum	249988686177578
Elapsed time	0.720119
MIPS	694
Accuracy	100

Conclusion and Future Work

Based on the outcome, we can see that multithreading or distributing the computations over multiple cores has a generous amount of performance benefits over single core/ single threaded operations provided it is done keeping resource allocations and deadlocks in mind. Also, based on the nature of the benchmark, it can be expected to be unbiased irrespective of platform. So, we can use it as a tool to measure compute ability of our systems

I have made minor but significant improvements in improving usability of my project by adding a GUI. While K-NN is not 100 percent tested, I expect it to be fully functional in an average use case. Most of minor improvements like use of batch files and joint execution of various benchmarks would go a long way in helping improve flexibility and modularity of my project. I will continue work on my project till it reaches point stable release for CPU and GPU cores.

Code

Added Code for reference:

```
#include<thread>
#include <omp.h>
#include <bits/stdc++.h>
#include <ctime>
#include <atomic>


void sumUp(std::atomic<unsigned long long> &sum, const std::vector<int> &v, unsigned long long start, unsigned long long end){
      unsigned int long long tmpSum{};
           for (auto i= start; i < end; ++i){
              tmpSum += v[i];
   }
              sum.fetch_add(tmpSum,std::memory_order_relaxed);
}
int main()
{
           //scrap
           /*size_t sum2;
           std::cout <<sizeof(sum);
           std::cout <<sizeof(sum2);
           printf("\nSIZE_MAX = %zu\n", SIZE_MAX);*/


           std::chrono::time_point<std::chrono::system_clock> start, end,start_1,end_1;
           std::default_random_engine rng;
           rng.seed(time(0));
           std::uniform_int_distribution<int> nd(0, 1000000);
           //unsigned long long sum = 0;
           unsigned long long sum_1 = 0;
           std::atomic<unsigned long long> sum(0);
```

```
        std::atomic<unsigned long long> sum2{};
            omp_lock_t writelock;
            omp_init_lock (&writelock);
            std::vector <int> v;
            for (int i = 0;i<500000000;i++)
                        {
                                    v.push_back(nd(rng));
                        }

            start_1 = std::chrono::system_clock::now();

            for (int i = 0;i<500000000;i++)
            {
             sum_1+=v[i];
            }
        end_1 = std::chrono::system_clock::now();

        std::cout <<"sum, sequential: "<<sum_1<<'\n';
    std::chrono::duration<double> elapsed_seconds_1 = end_1 - start_1;

    std::cout << "elapsed time: " << elapsed_seconds_1.count() << "s\n";
    float mops3 = 500000000/elapsed_seconds_1.count();
    std::cout <<mops3<<'\n';

//OpenMp
    start = std::chrono::system_clock::now();
#pragma omp parallel for
            for (auto i = 0;i<500000000;i++)
            {
                        sum.fetch_add(v[i]);
```

```
            }
        std::cout <<"sum: "<<sum<<'\n';

    end = std::chrono::system_clock::now();
    std::chrono::duration<double> elapsed_seconds = end - start;
    std::cout << "elapsed time for openmp parallelism: " << elapsed_seconds.count() << "s\n";
    float mops2 = 500000000/elapsed_seconds.count();
    std::cout <<mops2<<'\n';

//c++11
    auto start_2 = std::chrono::system_clock::now();
    std::thread t1(sumUp,std::ref(sum2),std::ref(v),0,125000000);
    std::thread t2(sumUp,std::ref(sum2),std::ref(v),125000000,250000000);
    std::thread t3(sumUp,std::ref(sum2),std::ref(v),250000000,375000000);
    std::thread t4(sumUp,std::ref(sum2),std::ref(v),375000000,500000000);

    t1.join();
    t2.join();
    t3.join();
    t4.join();
    std::chrono::duration<double> time_duration= std::chrono::system_clock::now() - start_2;
    std::cout << "Time for c++11 addition " << time_duration.count() << " seconds" << std::endl;
    std::cout << "Result: " << sum2 << std::endl;
    float mops = 500000000/time_duration.count();
    std::cout <<mops;
    std::cout << std::endl;
}
```

References

1. (https://www.webopedia.com/TERM/C/cpu-benchmark.html, n.d.)