Parallel computing assignment report

Vishal Keshav (11012341)

April 20, 2015

1 Convolution

1.1 Introduction

The problem is to compute the product of two polynomials given in the coefficient form.Coefficient can be anything from integer to decimal including complex numbers The basic $O(n^2)$ running time algorithm is surpassed by using discrete Fourier transform which has a running time of $O(nlog(n))$. Two sequential algorithms have been implemented, one with running time $O(n^2)$ and other with running time $O(nlog(n))$ and two parallel algorithms have been implemented for $O(nlog(n))$ version, one with OpenMP and other with C++11 threads. Comparison has been done for the running time of different algorithms. Programming is done in C++.

1.2 $O(nlog(n))$ Algorithm Pseudo code

```
Step 1: Input a and b(Coefficient of two polynomial)
Step 2: y1 = RecursiveFFT(a),y2=RecursiveFFT(b)
Step 3: y=y1*y2
Step 4: c = RecursiveFFTinv(y)/n;

RecursiveFFT(a):
n=a.length
w.n = $cos(2 * pi/n) + i * sin(2 * pi/n)$
w=1
a.even = a.even
a.odd = a.odd
y.even = RecursiveFFT(a.even)
y.odd = RecursiveFFT(a.odd)
for k=0 to n/2-1:
y[k] = y.even[k] + w*y.odd[k]
y[k+n/2] = y.even[k] - w*y.odd[k]
w = w*w.n
return y
```

RecursiveFFTinv(a):
(CHANGE in RecursiveFFT(a)): w.n = $cos(-2 * pi/n) + i * sin(-2 * pi/n)$

1.3 Theoretical Analysis

T.1(n) = 2*T.1(n/2) + O(n) = O(nlog(n))
T.inf(n) = T.inf(n/2) + O(n) = O(n)
Parallelism = O(n)

1.4 Program Analysis

The running time for higher input in case of sequential $O(n^2)$ and $O(nlog(n))$ is contrasting where $O(nlog(n))$ beats $O(n^2)$ at n^7. Parallel version gives a better performance than sequential $O(nlog(n))$ with O3 optimization. With openmp, performance is not good because of overhead.

1.5 Conclusion

From the observation from running time between optimized sequential and parallel code, it can be concluded that paralleling is good in the scenario where there are many physical processors available.

2 DFA

2.1 Introduction

Given a DFA and an input,problem was to make use of parallel processors to evaluate if the input is accepted by DFA or not. Programming was done on CUDA architecture with c-programming.

2.2 Algorithm

DFA transition matrix is made available to every processor and each processors were given a segment of input to process. Assuming states are comparatively low as compared to input length, each processor computes every possible state transitions with the input segments provided to it. These transition vectors present with every input were then reduced. Reduction is done in two ways: linear with $O(p)$ and with $O(log(p))$. p is the number of processors.

2.3 Theoretical analysis

Linear reduction: T(n) = $O(\frac{Qn}{p}) + O(Qp)$
For optimal performance, $\frac{Qn}{p} = Qpk$, gives
p = $O(\sqrt{n})$

then T(n) = $O(Q\sqrt{n})$
Log reduction: T(n) = $O(\frac{Qn}{p}) + O(Qlog(p))$
For optimal performance, $\frac{Qn}{p} = Qlog(p)k$, gives
p = $O(\frac{n}{log(n)})$
then T(n) = $O(Qlog(n))$

2.4 conclusion

With input of size n^{20}, parallel code given output in time half of that of sequential version.

3 Barrier up and in option pricing

3.1 Introduction

I solved the problem of estimating the price of Barrier up and in option taking the advantage of CUDA multiprogramming architecture. For price estimation, one need to discretize the time domain and generate a sequence of normal random variable to generate one path. At the end, option value is computed considering the past events happened with the path which is basically crossing the barrier. We need to generate many such paths in order to estimate a better price.

3.2 Algorithm

We run the path generation algorithm with thousands of threads. When each thread is finished, we take the average by log reduction. For the generation of normal random number in CUDA, mersener twister algorithm was used with uses 256 thread per random generator.

3.3 Theoretical analysis

Sequential case:T(n) = O(n*S), where n is path lenght and S is number of simulation.
Parallel case: T(n) = O(n). with S number of threads.

3.4 Conclusion

With S = 2^{10}, we observe parallel algorithm is 200 times faster.