Name: Ashwin Raghav Mohan Ganesh
UVA Id: am2qa

Assignment 1 -- CS 6444 -- Sequential Program Optimization

1.0 Introduction:

The challenge is to optimize a sequential program and gather run-time stats from the PBS cluster. Also, it is noted that the opimization is to be made to the sequential program without refactoring and making a threaded version. The objective of the exercise is to understand the operations of the GCC compiler and how these translate into the performance of an application. Another objective is to use the PBS cluster and understand the operations that can be performed on it.

2.0 Hardware and Software

Development Machines (2 configs tested):

2.1)
Dedicated personal Laptop
OS : Mac OSX Lion
Main Memory: 3GB DDR RAM
CPU: Intel x86-64bit I5

2.2)
Shared Department Systems (power1..6)
CS Department systems
OS :Ubuntu 10.04.3
Main Memory: 2GB DDR RAM
CPU: Intel x86-64bit XEON Quad Core

2.3)
Centurion001
CPU: AMD Opteron(tm) Processor 242
Main Memory 2GB RAM per node
OS :Ubuntu 10.04.3

Note: Disk/IO speed is inconsequential since the program has no CRUD operations.

3.0 Optimizations that worked well:

The code commit logs of these optimizations can be viewed incrementally in the online repository
https://github.com/ashwinraghav/Assignments/commits/master

Some of these optimizations were mentioned in the lecture and some others were the conclusions drawn from stats as shown by grpof.

3.01)

The exponentiation of the for e^a * e^b was clubbed as e^(a+b). This was the first step and I was immediately able to see a performance gain of 20%. However it is to be noted that this resulted in the precision of the resulting value being changed in the 7th decimal place (361030028.7871981263 to 361030028.7871980071). A reasonable trade-off for out particular case.

3.02)

Removed the inner loop division by ‘a’ and replace it with multiplying a constant ‘1/a’ value computed outside the loop. This converted n division operations to one division operation and n multiplication operations.

3.03)

Removed a branch in the second loop by running it till counter is ‘i’ instead of natom. This not only algorithmically reduces the time complexity (not asymptotically though) but also removes the inner condition that is crucial to let the compiler vectorize loops.

3.04)

Moved pow(cut,2) outside the loop. Brings the number of pow operations from n to 1.

3.05)

Some (3 coords[i] values and one q[i] value) array lookups were being performed on the inner loop for index values that were *invariant* inside the loop. These were moved outside to reduce the number of array lookups from 4*i to 4.

3.1 Some significant optimization lessons:

3.06)

One of the most optimizing moves was to split the second loop into 2 separate loops. One to computer Vector values and the other to conditionally increment total_e. This meant that one loop containing the 3 pow operations was not vectorizable. This improved performance by 20%.

3.07)

Another optimization step was converting the 2-dimensional double type array into a one-dimensional array and manually keeping track of the array indices. The overhead of calculating the correct index clearly won over the cache misses that would be caused in a 2D array. There was a significant performance gain once again of 25%. This particularly made a difference since the original code had array lookups in a row major manner which causes a cache miss almost every time.

4.0 Failed Optimizations

The inverse square root computation that was recommended in the lecture did not work too well on all machines. It was noted infact that in both development environments and on the cluster there was nearly a 30% drop in performance when the division by square root was replaced by multiplication of the reciprocal's square root

4.1 Disappointment in AMD's vector library

4.11)

Since the cluster machines were observed to be AMD 64 bit machines, I was hopeful that **AMD 's libm** for optimized math operations will help increase performance. Inspite of compiling the library from source on centurion001, performance only seemed to drop by 30% when these libraries were used. Quite ironically the resulting binaries were faster on my development Intel processors as opposed to AMD's own in Centurion001.

4.12)

Converting the 3rd loop contents into a function invoked through a pointer did not vectorize it. Intuitively it seemed as though changing the branched out code into a function would be a way to cheat the compiler into not seeing the branch and vectorizing the loop. However, GCC seemed to recognise pointer aliases and failed to vectorize the 3rd loop. A feature that actually turned out to be a bug.

5.0 Performance Statistics

In the interest of brevity I have provided stats for power1 development box (no 2 in the list) and Centurion001(no 3) and skipped measuring running time with flags on my local machine(no1).

The following flags that that i narrowed down on based on intuition and not stats.

Flag	Description
ffast-math	Turned on by default for all -O options
ftree-vectorizer-verbose=5	Displays unused variables and vectorization guidelines -- highly useful to improve performance in incremental steps.
-m64	use 64 bit registers
msse2	Since AMD's opteron and Intel's Xeon support sse.

-march=native	native ARM architecture
-mtune=native	tune the generated instructions to native ARM

5.1 Intermediate Readings Gathered on development machine

All intermediate readings were taken from power1.cs.virginia.edu
All Flags were used and optimization level was O3.

Optimization Number	**Total Execution Time (s)**	**Carried over to final set of optimizations**
without manual optimization	12.4900	Not Applicable
3.01	10.2800	yes
3.02	9.4800	yes
3.03	8.4500	yes
3.04	8.1400	yes
3.05	7.8500	yes
3.06	7.5400	yes
3.07	7.4300	yes
4.11	33.32	NO
4.12	7.51	NO

5.2 PBS config

```
#!/bin/sh
#PBS -l nodes=1:ppn=1
#PBS -l walltime=12:00:00
#PBS -o output.txt
#PBS -j oe
#PBS -m ea
#PBS -M am2qa@virginia.edu
```

5.3 Fully Optimized Readings

5.31 Centurion001--after complete manual optimization

Optimization Level	**Time to Calculate E**	**Total Execution**
3	12.3600	12.4100
2	12.3500	12.4100
1	13.2100	13.2600
no compiler optimization	20.59	20.64
no flags	21.12	21.18

5.32 power1.cs.virginia.edu

Optimization Level	**Time to calculate E**	**Total Exectution**
3	7.5400	7.5700
2	7.5800	7.6100
1	8.0100	8.0400
no flags	8.24	8.27
no compiler optimization	11.800	11.8400

6.0 Conclusion:

An important lesson I have learnt out of this exercise is that software ‘s performance must first be tuned to run well sequentially before parallelization. There are methods other than parallelization and algorithmic optimizations that can help improve the running speed of software. Also, it is important to be able to intuitively and non-heuristically be able to tell a compiler how a piece of software must be optimized on account of the number of options out there. In-depth knowledge about the target platform can help these decisions.

Another important lesson out of this exercise was that, although there are libraries out there that are vectorised/optimized, they need to be judged and used on a case by case basis. Extensive testing of the performance is a must on the target platform.

6.1 Flags used:

It is evident from the statistics that the performance drops with the level of optimization and with the usage of flags. I believe that this is indicative of the fact that the flags used were appropriate and in accordance to the specs of the target platform.
Also, it is to be noted that since no parallelization was adopted, running the software on the cluster did not result in a performace gain. In fact there was a performance drop on account of the individual CPU s being less performent than the machines in the development environment.