CS-301
High Performance Computing

Assignment 2

Amarnath Karthi 201501005
Chahak Mehta 201501422

August 28, 2017

1 Hardware Details

Parameter	Value
CPU model	Intel Core i5-4590 3.30 GHz
Number of Cores	4
L1d Cache	32 kB
L1i Cache	32 kB
L2 Cache	256 kB
L3 Cache	6144 kB
Compiler	gcc

2 Problem Statement

Write and analyze serial code and parallel code for the following using OpenMP.

1. **Integration of a function using trapezoidal rule. Verify your code by using it to calculate the value of π.**
2. **Calculation of π using series.**
3. **Summation of two vectors**

3 Solution

3.1 Integration using trapezoidal rule

3.1.1 Theory

We use the following formula to compute the value of π :

$$\int_0^1 \frac{4}{1+x^2} dx \tag{1}$$

3.1.2 Approach

For the serial code, we use simple numerical integration over 0 to 1. For parallel implementation, we simply run numerical integration over 4 equal ranges of length 0.25 between 0 and 1 over 4 parallel threads. We add each partial sum to the global sum, taking care that the global addition takes place under the **critical** section.

3.1.3 Results and Observations

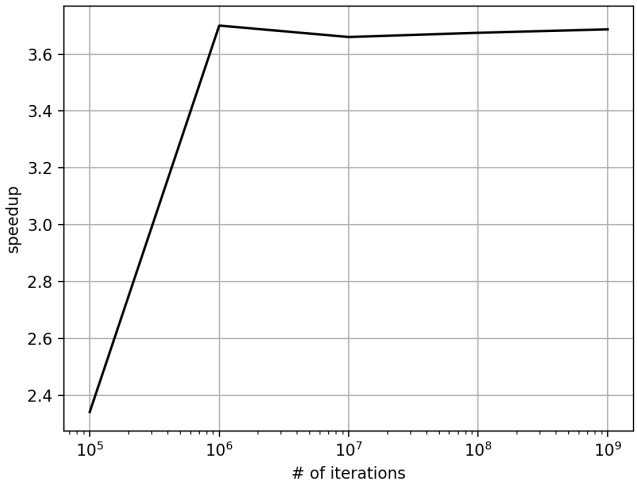

Figure 1: speedup vs number of iterations

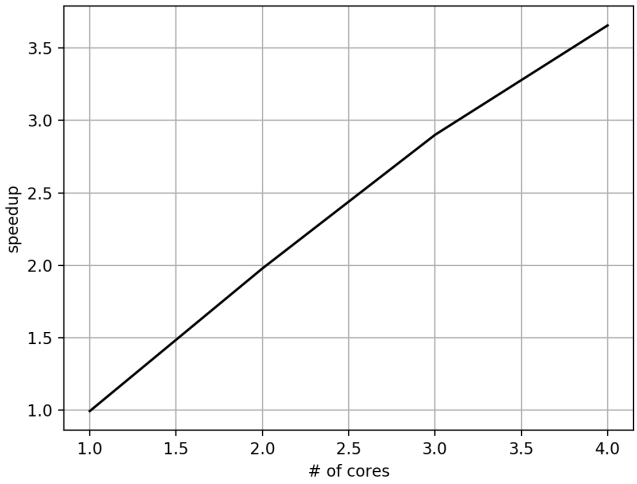

Figure 2: speedup vs number of cores

The above plots show speedup as a function of number of iterations (the problem size) and the number of cores respectively. As the computing unit used during the experiment had 4 cores, the problem could be broken down into 4 parallel processes each having an equal problem size. Hence roughly the speedup should have been at most 4. This is in agreement to our observations which give us a maximum speedup of 3.73.

In **figure 2** we see speedup increasing as the number of cores increase. This is in agreement with **Amdahl's law** which gives us a rough upper limit on speedup in terms of parallelization and also gives us speedup as a function of ratio of the parallel code to serial code and the number of cores in a machine. For a piece of code having a ratio s as the serial part and p as a parallel part running on a machine having n cores, the maximum theoretical speedup which can be received is given by :

$$speedup = \frac{1}{s + \frac{p}{n}} \tag{2}$$

From **figure 1** we see the benefits of parallelization for large input sizes. For small input sizes of the order 10^5 the parallel code is about 2.4 times faster than its serial counterpart. When the input size grows to 10^7, the speedup increases by nearly 50%.

4 Coding strategies

Implement some of the important optimization strategies discussed during the lecture today and measure difference in improvement while using the best strategy.

1. **Case A**

```
for (int i = 0; i < N ; i ++)
    if( A [ i ] > 100) {
        flag = true ;
        break ;
    }
```

Case B

```
for (int i = 0; i < N ; i ++)
    if( A [ i ] > 100) {
        flag = true ;
    }
```

Observation: When N = 1×10^9 in the above cases, and we search in an already instantiated array, the flag is turned true when the value of A[i] is greater than 100 which in this case was kept at i = 500000000. Theoretically, case B does all the iterations while case A has a break condition if the if condition returns true. Case A takes 0.989066 seconds while case B takes 1.956732 seconds.

Optimization strategy: *Trying to perform the least number of operations if the output remains same.*

2. **Case A**

```
A = B**2.0;
```

Case B

```
A = B*B
```

Observation: For large floating point numbers of the order 1×10^{25}, Case A takes 0.000019 seconds while case B takes 0.000000000001 seconds.

Optimization strategy: *Try and avoid more resource consuming operations. In the first case, power does multiplication O(log(n)) times while in the second case, multiplication is done only one time.*

3. **Case A**

```
int s = 1 , r =2;
for (int i = 0; i < N ; i ++) {
    A [ i ] = A [ i ] + s + r * sin (0.523598776) ;
}
```

Case B

```
int s = 1 , r =2;
double tmp = s + r * sin (0.523598776) ;
for (int i = 0; i < N ; i ++) {
    A [ i ] = A [ i ] + tmp ;
}
```

Observation: As we can see in the code, the operation $s + rsin(x)$ takes place 10^8 times in case A while it takes place only once in case B which makes it faster. Case A takes 3.584542 seconds, Case B takes 2.890220 seconds.

Optimization strategy: *Reduce the number of registers used to store data, the number of comparisons made or the number of redundant operations*

4. **Case A**

```
Branched matrix multiplication where N = 1x10000
```

Case B

```
Unbranched matrix multiplication where N = 1x10000
```

Observation: Case A takes 0.401576 seconds while case B takes 0.194602 seconds. The chances of a branch miss is higher in A then in B because of more number of branches in A than in B. This increases the latency and hence takes more time.

Optimization strategy: *Avoid branching wherever possible.*