Assignment 3 **Akshat Tandon**
CS4850 Performance Evaluation of Computer Systems 201503001

Performance Metrics and Measurement

Execution Time

a) Wall clock time versus CPU time

i)

Assumption The wall clock and CPU time are calculated by considering only the matrix multiplication function and not the entire program consisting of matrix declarations and other overheads.

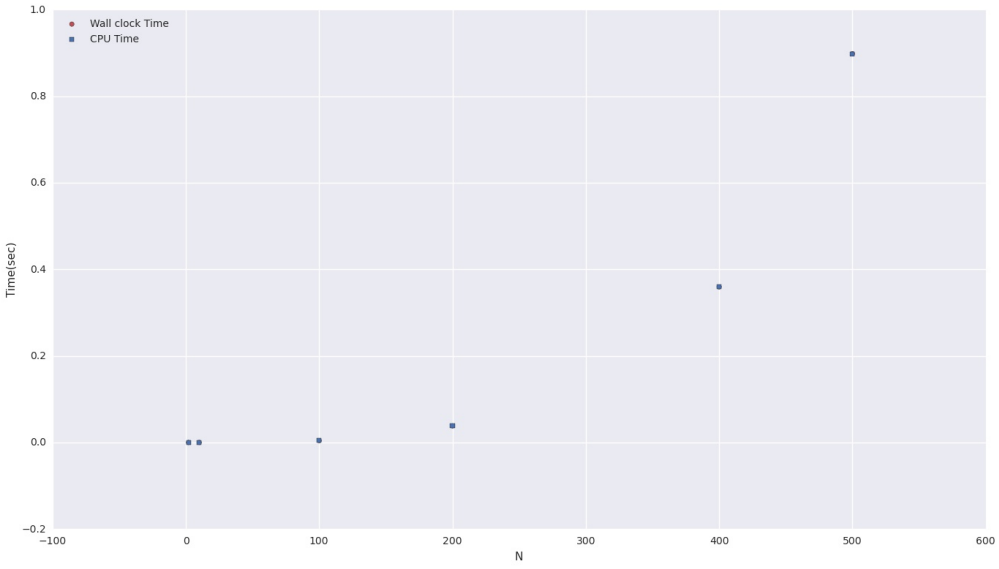

Figure 1: Wall-clock and CPU time for N = 2, 10, 100, 200, 400, 500

Comments

- The difference between wall-clock and CPU time is very minuscule(at the scale of 10^{-5} to 10^{-3}).
- Due to this minuscule difference, the wall-clock and CPU time points look as if they are overlapping(figure 2). One needs to look at a much higher scale in order to differentiate between the two types of points(figure 3).
- Both the times are monotonically increasing with N, till N = 200(figure 2). Beyond that, there are a few irregularities in which $t_{n1} > t_{n2}$ but $n1 < n2$. One possible explanation for this anomaly appears that as we increase N, the time spent by the program in CPU increases. As more time is spent, there is a greater chance that the multiplication program

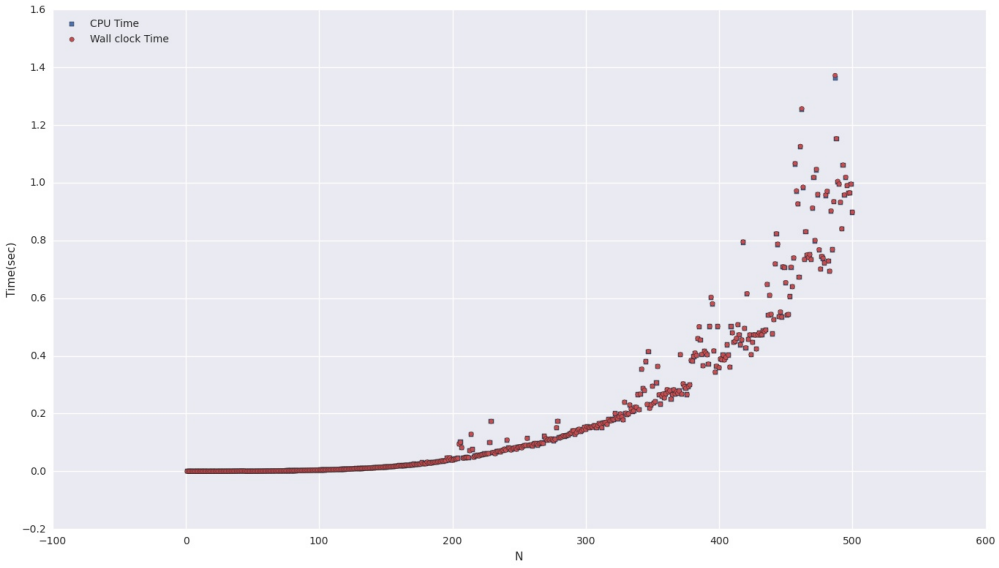

Figure 2: Wall-clock and CPU time for N ranging from 1 to 500

might be stalled by the CPU so as to process some other kernel or user program. Due to these sudden and erratic interrupts caused due to other programs, inspite of having a lower N, the time spent is more.

Key Differences

- CPU time is always less than the wall-clock time in every instance.

(ii)

Wall clock time It is the time elapsed since the block of code under consideration started to execute till the time it stopped. It includes the time spent by the block of code in the CPU as well as the time during which the code block stays stalled while other user or kernel processes consume the CPU. It also includes Input/Output related stalls as well as stalls due to any other system overheads.

CPU Time It is the time spent by the block of code under consideration in the CPU. It is only concerned with the the time which the code block spends in the CPU and does not regard the time during which the code block remains stalled due to other processes. It does not include any stalls due to Input/Output or system overheads.

Considering the above explanations for both types of times,

- Wall clock time should be used in scenarios in which we want to measure how a given program will perform in the presence of other programs, in a generalized kind of setting. It should be used when we want to analyze the program's performance under various kinds of system and user overheads. Also, in scenarios when the Input/Output forms a significant part of the program, wall clock time should be considered.

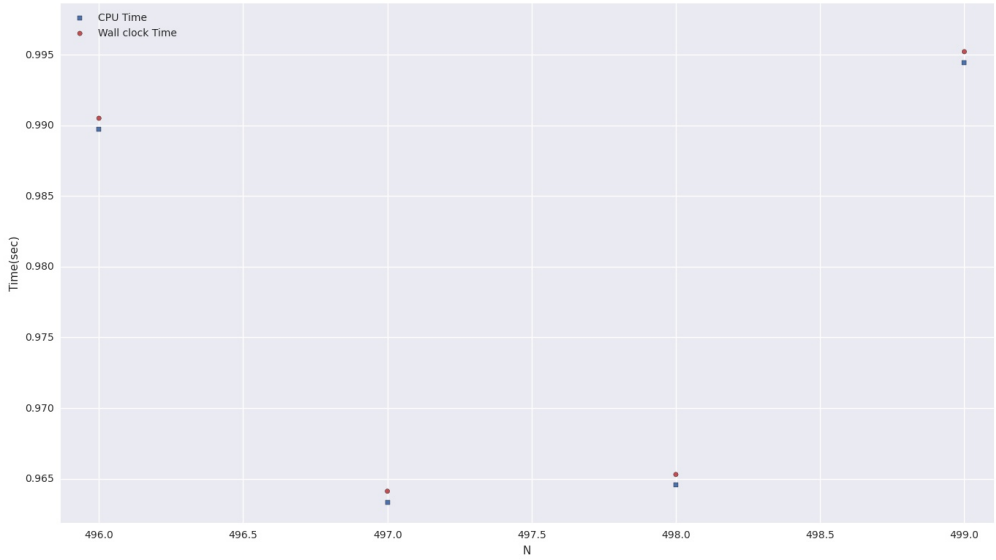

Figure 3: An enlarged view of the CPU and Wall-clock time graphs

- CPU time should be used when we are concerned with the standalone performance of the process. It should be used when we want to analyze only the program under consideration without any outside interference caused due to kernel or other user programs.

Generally, the wall clock time is greater than the CPU time. However, it is possible that both the times turn out to be same for a certain program run under a certain situation. This happens only when during the process execution, the process does not get stalled by the CPU so as to run some other kernel or user process or due to some I/O operation.

iii)

	Now	10 mins later	20 mins later
Wall-clock time(sec)	0.569241	0.581752	0.553329
CPU Time(sec)	0.568844	0.581331	0.552908

Table 1: Wall clock and CPU times for N = 450 at various time instances

Comments The wall clock and CPU time at all the three time instances come out to be different.

Why is wall clock time different ? It is easy to see why the wall clock time would be different. The wall clock time depends on external influences such as other user/kernel process'. On a modern , machine, no program is run in isolation and there are always some background user and kernel process' running. These background process' can preempt the matrix process at different time steps. Since this manner of preemption is quite erratic and stochastic, the measured wall time is different at different time steps.

Why is CPU time different ? Now this seems to be an anomaly. Since the CPU time does not depend on external influences, it should come out to be the same at each time instant. However there are two reasons for this anomalous behavior

- Since N = 450 and *long long* data type is used to store an array element, the total size occupied by the array is (number of elements) * (size of long long) = (450*450)*(8) = 1620000 bytes = 1.54 MB. The cache of my systems CPU is 3MB. Due to the presence of several background process' and the high data to cache size ratio(1:2), a high number of data cache misses take place. These data cache misses stall the currently executing matrix multiplication process even though the multiplication process has not been prempted by some other process and is still in the CPU backend. Since the multiplication process has not been prempted by the CPU core, this time is added to the CPU time. This stall time due to data cache misses is variable and causes variability in CPU time measured at different time instants.
- The CPU time is measured using the function **clock()** defined in *time.h*. According to the standard c reference, this function
  "Returns the approximate processor time used by the process since the beginning of an implementation-defined era related to the program's execution.". This approximation is also responsible for the variability in the CPU times.

iv)

Due to the variability in both the wall clock and CPU times, we should measure both of these times at several different time instants(3 time instants are not sufficient, a large number of samples are needed so as to obey the law of large numbers) throughout the day, and then find the mean of these different measurements. Mean is suitable for summarizing the wall clock and CPU times since these values are neither categorical nor is their distribution much skewed. These means should be reported as the wall-clock and CPU times for the respective program. If we are given only the above three measured times and are not able to get many samples, then we can give the mean of those three particular values resulting in the mean wall clock time: $\frac{0.569241+0.581752+0.553329}{3} = 0.5681sec$ and mean CPU time: $\frac{0.568844+0.581331+0.552908}{3} = 0.56769sec$

b) MIPS versus MFLOPS

i)

Deriving MIPS The key step in the matrix multiplication program, which is repeated again and again is

```
c[i][j] += a[i][k]*b[k][j];
```

This single C instruction approximately corresponds to the following x86 assembly

```
movl -12(%rbp), %eax
imull -8(%rbp), %eax
addl %eax, -4(%rbp)
popq %rbp
```

Thus it is safe to assume that calculating a new element of the product array requires 4 x86

instructions. Now for computing the product of two $N \times N$ matrices, N^3 such C instructions or $4 \times N^3$ x86 instructions are executed. Therefore

$$MIPS = \frac{4 \times N^3}{cpu - time_n} \times 10^{-6} \tag{1}$$

where N is the dimension of the matrix and $CPU - time_n$ is the CPU time corresponding to that value of N.

Deriving MFLOPS Again consider the key step in C program required for calculating the elements of the product matrix
`c[i][j] += a[i][k]*b[k][j];`
If `a[i][k]` and `b[k][j]` are floating point numbers, then this key step involved 2 floating point operations, one floating point addition and one floating point multiplication. For computing the product of two $N \times N$ matrices, N^3 such C instructions are required which implies that $2 \times N^3$ floating point operations are performed. Therefore

$$MFLOPS = \frac{2 \times N^3}{cpu - time_n} \times 10^{-6} \tag{2}$$

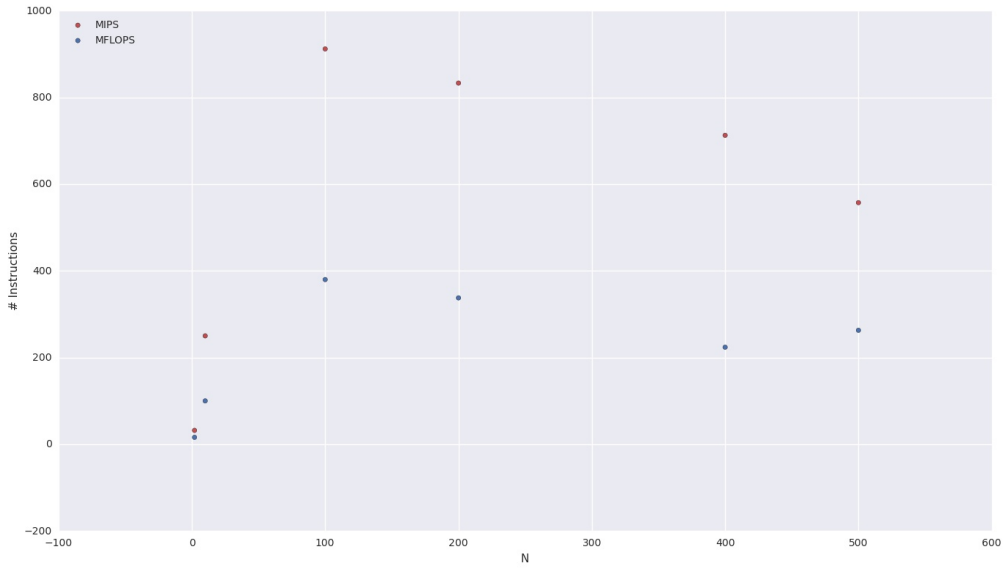

Figure 4: MIPS and MFLOPS when N = 2, 10, 100, 200, 400, 500

Comments

- The general trend for both MIPS and MFLOPS metric is first increasing (till around N = 100) and then decreasing(with increasing N).
- MIPS is greater than MFLOPS for nearly all values of N. This is due to two reasons. The first major reason is that MIPS is greater than MFLOPS by a factor of 2 (see the above formulas). The second reason is that CPU time in case of floating point numbers is greater when compared to integral numbers.

Figure 5: MIPS and MFLOPS for N ranging from 1 to 500

ii)

The CPU in my system is Intel(R) Core(TM) i5-3337U CPU which operates at a base clock frequency of 1.80GHz. Though it is difficult to estimate the IPC(Instructions Per Cycle) of this particular CPU, but IvyBridge CPU's(family name of my systems CPU) on an average can decode upto 3 CISC instructions per cycle. Under this assumption, the average MIPS processed by this CPUS is $1.80 \times 10^3 \times 3 = 5400$.
The average MIPS for N ranging from 1 to 500 for the matrix multiplication program is 685.517. Thus the MIPS calculated using assembly instruction approximations and CPU times is smaller then the one approximated using IPC and base frequency by a factor of $\frac{5400}{685.517} = 7.877$.

Perf Linux Profiling with Performance Counters

2

a) Time

Comments

1. The difference between all the four types of time is miniscule due to which all the four time points for a particular N appear overlapping.

2. For all N, perf metric1 > perf metric2 > wall clock time > CPU time.

Perf command explanation

1. The **time-elapsed**(metric1) measures the wall clock time elapsed from the point the program started to execute till the time it stopped executing.

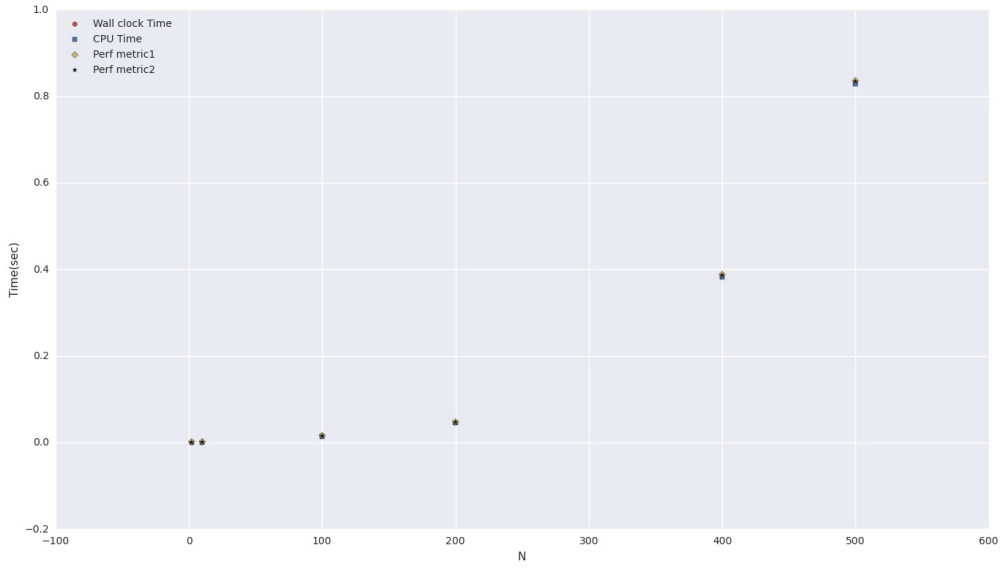

Figure 6: Perf time metric 1 and 2, wall clock time and CPU time for N = 2, 10, 100, 200, 400, 500

2. The time calculated by diving the number of cycles with the frequency gives the time spent by the entire program in the CPU. This also includes the time during which the program remains stalled in the CPU due to instruction and data cache misses.

Comparison of perf times with the above measured wall clock and CPU times

1. The key difference between the perf times and our measured wall clock and CPU times is that the perf times consider the time duration of the entire program whereas our measured wall clock and CPU times only considers the time spent/elapsed only by the matrix multiplication function of the program.

2. Since the perf times considers the entire program as opposed to just the matrix multiplication operations, it considers the time spent to create/allocate the matrices whose multiplication has to be performed as well as other overheads. Thus **metric1** and **metric2** are always greater than our wall clock and CPU times.

3. For smaller values of N(2 or 10), the wall time and CPU time are much smaller as compared to their perf counterparts(see the ratios in Table2), however this difference decreases as N increases. This is because when N is small, the program overheads like array declaration overpower the small number of add/multiply operations but as N increases these add/multiply operations overpower the program overhead.

b) Number of Instructions

Comments

1. Instruction metric 1 and 2 are monotonically increasing.

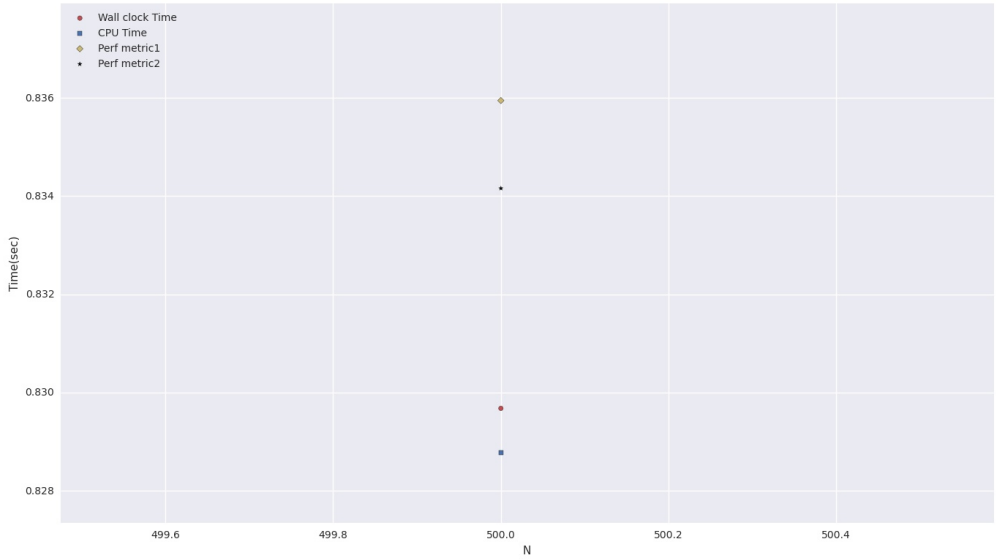

Figure 7: Enlarged view of the graph showing Perf time metric 1 and 2, wall clock time and CPU time

	N: 2	10	100	200	400	500
Wall-clock time(s)	0.000005	0.000028	0.014159	0.045424	0.382602	0.829677
CPU Time(s)	0.000001	0.000021	0.014073	0.045272	0.381568	0.828776
perf time 1(s)	0.0015732	0.0018755	0.0162519	0.0477056	0.3882094	0.8359442
perf time 2(s)	0.000509071	0.000963615	0.01526203	0.0469210	0.386577	0.83415
$\frac{wall-time}{perf-1}$	0.00317807	0.01492882	0.87121682	0.95217279	0.98555569	0.99250276
$\frac{CPU-time}{perf-2}$	0.00196436	0.02179293	0.92209218	0.96485474	0.98704128	0.99355033

Table 2: Wall clock, CPU, metric 1, metric 2 times and their associated ratios for varying N

2. Instruction metric 3 remained fairly constant.
3. Metric 1 > Metric 2 > Metric 3 for all N.
4. The difference between metric 1 and 2 is very small.

Explaining perf instruction metrics

1. Instruction metric 1 denotes the total number of x86 assembly instructions executed for the user(the matrix multiplication process) as well kernel processes responsible for executing the matrix multiplication code.
2. Instruction metric 2 denotes the total number of x86 assembly instructions executed only for the user process'(the matrix multiplication process) responsible for executing the matrix multiplication code.
3. Since metric 1 also includes the kernel instructions executed, it is always greater than metric 2 which only includes the instructions of the program itself.

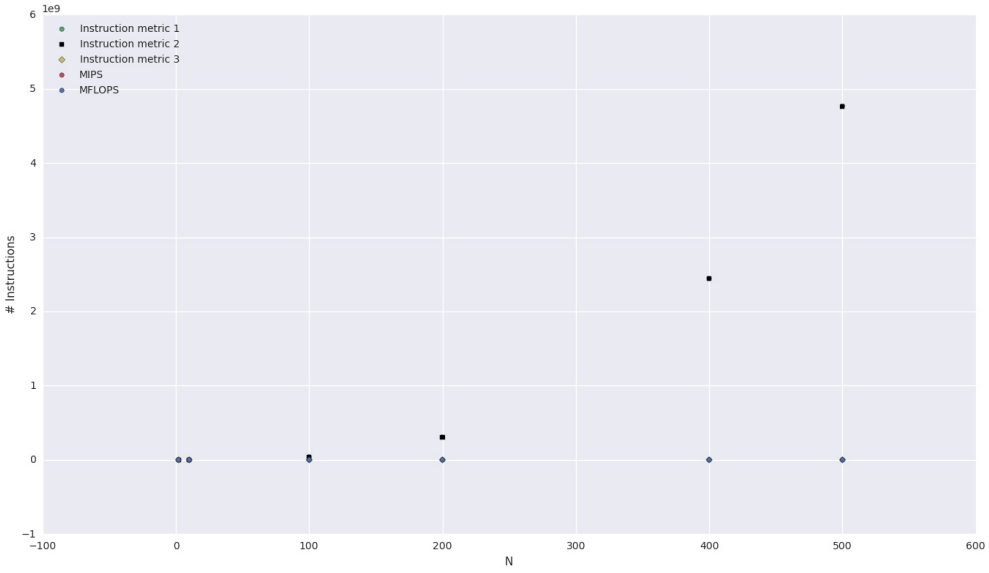

Figure 8: Perf instruction metric 1, 2, 3 and MIPS, MFLOPS for varying N

4. On increasing N, the number of assembly instructions for the user process(the matrix multiplication process) increases since more number of addition and multiplication operations are required. Due to this on increasing N, both metric 1 and 2 increase.

5. Instruction metric 3 denotes the number of single precision floating point operations executed by the user process. This metric remains fairly constant in the range of around 450-500 and does not vary much on changing N. This was because during the experiments, only integral matrices were considered.

6. On considering floating point matrices, metric 3 monotonically increases with increasing N but still always remains less than metric 1 and 2 since metrics 1 and 2 include the floating point type instructions in addition to other instructions.

Contrasting perf instruction metrics, MIPS and MFLOPS

We calculated the per unit cpu-time counterparts of perf instruction metrics 1,2 and 3 so as to compare them with MIPS and MFLOPS.

$$MM1IPS = \frac{instructionMetric1 \times 10^{-6}}{cpuTime} \tag{3}$$

$$MM2IPS = \frac{instructionMetric2 \times 10^{-6}}{cpuTime} \tag{4}$$

$$MM3IPS = \frac{instructionMetric3 \times 10^{-6}}{cpuTime} \tag{5}$$

Comments

- MM1IPS and MM2IPS are greater than all the other metrics.

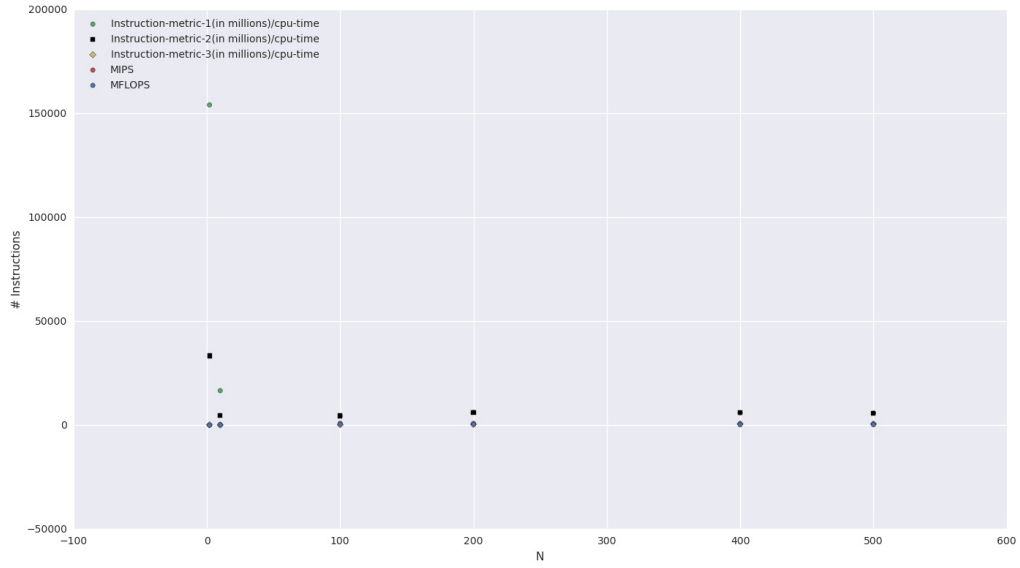

Figure 9: Perf instruction(in millions) metrics 1, 2, 3 per unit CPU time; MFLOPS ; MIPS for varying N

- On increasing N, MM1IPS and MM2IPS both converge to a similar value.
- As mentioned earlier, depending on the whether the matrices being multiplied contain floating point or integral numbers, instruction 3 metric will either be increasing or static. MM3IPS is calculated using floating point matrices so that it exhibits some variability.
- MM1IPS and MM2IPS for N=2 is abnormally high. It is because due to the small matrix dimension, a very small number of addition/multiplication are executed resulting in a very minuscule cpu-time. This minuscule cpu-time distorts the MM1IPS and MM2IPS values.
- Ignoring the anomalous behavior at N=2, MM2IPS is greater than MIPS by a factor of 10 - 20(see $\frac{MM2IPS}{MIPS}$ ratio values in Table3). This is because the MIPS we defined in previous sections only considers 4 assembly instructions associated with the addition and multiplication operations, per update, whereas MM2IPS considers the assembly instructions which make up the entire matrix multiplication program.
- Ignoring the anomalous behavior at N=2, MM3IPS is greater than MFLOPS by a factor of 0.8 to 2.6(see $\frac{MM3IPS}{MFLOPS}$ ratio values in Table3). This is because the MFLOPS we defined in previous sections only considers one floating point addition and one floating point multiplication per update, whereas MM3IPS consider all the floating point operations occurring anywhere in the program.
- The CPU used during the analysis was an Intel(R) Core(TM) i5-3337U CPU @ 1.80GHz. It belongs to the ivy-bridge family and has the hex code rff10 for single precision floating point operations(same as sandy bridge).

	N: 2	10	100	200	400	500
CPU Time(s)	0.000003	0.000031	0.008619	0.049504598	0.407206	0.825030
$inst^n metric1$	462358	515108	39110211	307068533	2443024104	4767206620
$inst^n metric2$	100162	142228	38561736	305944102	2439469178	4761612090
$inst^n metric3(f)$	578	2652	2111379	20566310	241973924	469496181
$inst^n metric3(i)$	548	587	525	670	550	547
MM1IPS	154119.333	16616.3870	4537.67386	6202.82853	5999.47963	5778.2221
MM2IPS	33387.3333	4588.0	4474.0382	6180.11486	5990.74959	5771.44114
MM3IPS	192.6666	85.54838	244.9679	415.4424	594.2297	569.0655
MIPS	32.0	250.0	911.78481	833.0296	712.554	557.2390
MFLOPS	16.0	100.0	379.867046	337.60972	223.800873	262.872055
$\frac{MM2IPS}{MIPS}$	1043.35416	18.352	4.90690149	7.41884163874	8.40743201	10.3572089
$\frac{MM3IPS}{MFLOPS}$	12.0416666	0.855483870	0.644878201	1.23054044	2.65517177	2.16480060

Table 3: Various types of perf instruction metrics as well as instruction per second metrics for varying values of N. Note that $inst^n metric3(f)$ is calculated using floating point matrices and $inst^n metric3(i)$ is calculated using integral matrices.

Matrix multiplication C code

Two separate C codes were used for integral and floating point matrices respectively. In the floating point case, the only change was made to the data type of the matrix, which was changed from `long long` to `double`.

0.1 Code

```c
#include <stdio.h>
#include <time.h>
#include <sys/time.h>

long long N;

double currentWallClockTime()
{
  struct timeval time;
  if(gettimeofday(&time, NULL))
    return 0;
  return (double)time.tv_sec + (double)time.tv_usec * 0.000001;
}

double currentCPUTime()
{
  return (double)clock()/CLOCKS_PER_SEC;
}

void multiplyMatrix(long long a[N][N], long long b[N][N], long long c[N][N])
{
  long long i, j, k;
  for(i = 0; i < N; ++i){
    for(j = 0; j < N; ++j){
      c[i][j] = 0;
      for(k = 0; k < N; ++k){
        c[i][j] += a[i][k] * b[k][j];
      }
```

```
    }
  }
}

void printMatrix(long long a[N][N])
{
  int i, j;
  for(i = 0; i < N; ++i){
    for(j = 0; j < N; ++j){
      printf("%lld | ", a[i][j]);
    }
    printf("\n");
  }
  printf("\n");
}

int main()
{
  int MAX_N = 10;

  double wall_clock_times[MAX_N];
  double cpu_times[MAX_N];

  int idx, i, j;
  long long counter;

  FILE *f = fopen("wall_cpu_times_temp.txt", "w");
  if (f == NULL)
  {
      printf("Error opening file!\n");
      exit(1);
  }
  fprintf(f, "N,wall_clock_time,cpu_time\n");
  for(idx = 1; idx <= MAX_N; ++idx)
  {
    N = idx;

    long long a[N][N];
    long long b[N][N];
    long long c[N][N];

    counter = 0;
    for(i = 0; i < N; ++i){
      for(j = 0; j < N; ++j){
        a[i][j] = counter;
        b[i][j] = counter;
        counter++;
      }
    }
    // printMatrix(a);
    // printMatrix(b);

    double wall_clock_tick = currentWallClockTime();
    double cpu_tick = currentCPUTime();

    multiplyMatrix(a,b,c);

    double wall_clock_tock = currentWallClockTime();
    double cpu_tock = currentCPUTime();

```

```
89      // printMatrix(c);
90
91      wall_clock_times[idx-1] = wall_clock_tock - wall_clock_tick;
92      cpu_times[idx-1] = cpu_tock - cpu_tick;
93      // printf("Idx: %d", idx);
94      // printf("Wall clock time: %lf \n", wall_clock_times[idx - 1]);
95      // printf("CPU time: %lf \n", cpu_times[idx - 1]);
96      fprintf(f, "%d,%lf,%lf\n", idx, wall_clock_times[idx - 1], cpu_times[idx -
        1]);
97    }
98    fclose(f);
99
100
101   // printMatrix(c);
102   return 0;
103 }
```

Readme

The above program creates a file containing the wall clock and CPU time for matrices ranging in size from 1 to MAX-N. This MAX-N can be changed accordingly in the main() function of the program.

Compiler options used

To compile the program, the following command was used
`gcc -Wall -o2 matrix.c`