

Timing Programs and Performance Analysis

Tools for Analysing and Optimising
advanced Simulations

Performance Analysis

When analysing the performance of a program, there are a number of aspects that can be measured/considered:

- Execution time
- CPU utilisation
- Memory usage
- Disk usage
- Bandwidth
- Power consumption

Execution Time

Even a simple aspect such as the execution time of a program can be difficult to measure or even define.

By execution time are we measuring:

- Wall clock time - total elapsed time.
- CPU time (or process time) – amount of time that the CPU spent executing that program.

What if we want to measure the execution of only one part of the program?

Measuring Execution Time

There are a number of approaches for measuring execution time in a program:

Bash `time`:

```
$time    ./myprogram
real    0m0.00s
user    0m0.00s
sys     0m0.00s
```

Measuring Execution Time

Programs to profile:

Row Major

```
// Memory allocation
unsigned int *p = new unsigned int[N*N];
// Initialisation
for(int y = 0; y < N; y++) {
    for(int x = 0; x < N; x++) {
        p[y*N + x] = x*y;
    }
}
// Initialisation
unsigned int total = 0;
for(int y = 0; y < N; y++) {
    for(int x = 0; x < N; x++) {
        total += p[y*N + x];
    }
}
cout << "Total is: " << total << endl;
delete[] p;
```

Column Major

```
// Memory allocation
unsigned int *p = new unsigned int[N*N];
// Initialisation
for(int x = 0; x < N; x++) {
    for(int y = 0; y < N; y++) {
        p[y*N + x] = x*y;
    }
}
// Initialisation
unsigned int total = 0;
for(int x = 0; x < N; x++) {
    for(int y = 0; y < N; y++) {
        total += p[y*N + x];
    }
}
cout << "Total is: " << total << endl;
delete[] p;
```

Measuring Execution Time

Row Major (N=1024)

```
$ g++ row-major.cpp -o row-major  
$ time ./row-major  
Total is: 3758368528
```

```
real    0m0.015s  
user    0m0.008s  
sys     0m0.004s
```

```
$ g++ row-major.cpp -o row-major -O3  
$ time ./row-major  
Total is: 3758368528
```

```
real    0m0.009s  
user    0m0.003s  
sys     0m0.004s
```

Column Major (N=1024)

```
$ g++ col-major.cpp -o col-major  
$ time ./col-major  
Total is: 3758368528
```

```
real    0m0.033s  
user    0m0.026s  
sys     0m0.004s
```

```
$ g++ col-major.cpp -o col-major -O3  
$ time ./col-major  
Total is: 3758368528
```

```
real    0m0.025s  
user    0m0.019s  
sys     0m0.004s
```

Measuring Execution Time

Row Major (N=8192)

```
$ g++ row-major.cpp -o row-major  
$ time ./row-major  
Total is: 16777216
```

```
real    0m0.601s  
user    0m0.465s  
sys     0m0.134s
```

```
$ g++ row-major.cpp -o row-major -O3  
$ time ./row-major  
Total is: 16777216
```

```
real    0m0.281s  
user    0m0.139s  
sys     0m0.140s
```

Column Major (N=8192)

```
$ g++ col-major.cpp -o col-major  
$ time ./col-major  
Total is: 16777216
```

```
real    0m7.579s  
user    0m7.435s  
sys     0m0.140s
```

```
$ g++ col-major.cpp -o col-major -O3  
$ time ./col-major  
Total is: 16777216
```

```
real    0m2.200s  
user    0m2.051s  
sys     0m0.147s
```

Measuring Execution Time

The wall clock or real time is the actual elapsed time of your program.

System time is the time spent doing system services.

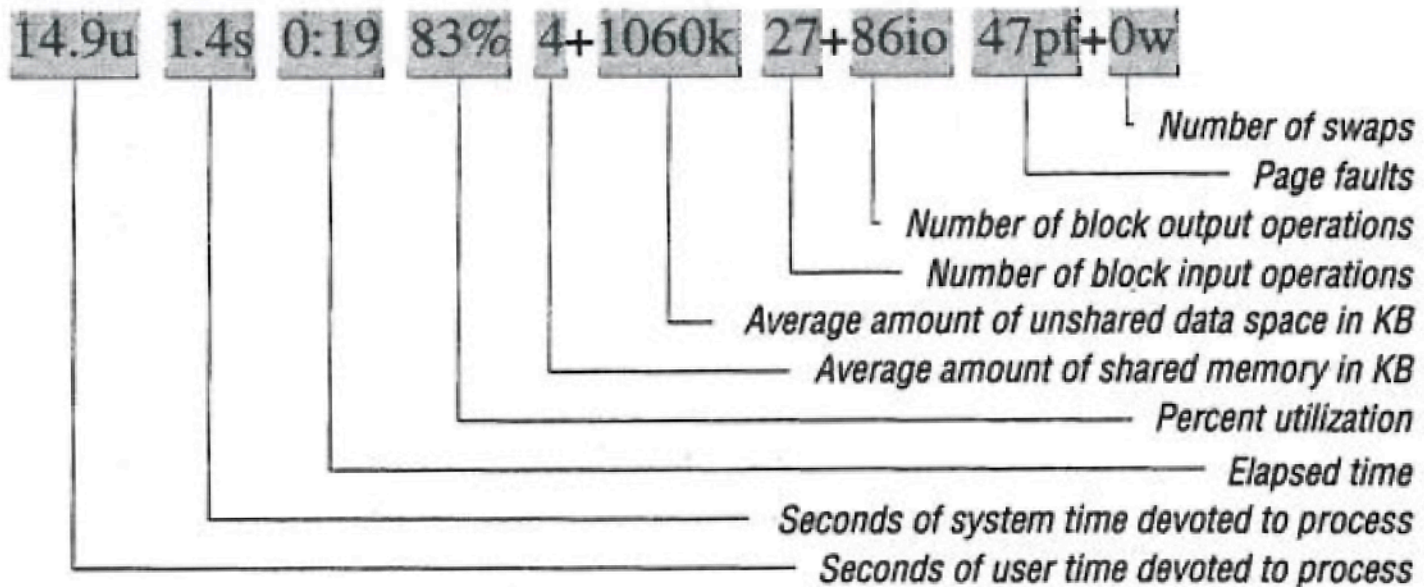
User time is the time your program was actually running.

$$\text{Real Time} \approx \text{System Time} + \text{User Time}$$

Measuring Execution Time

The `time` utility on `csh` or `tcsh` gives somewhat more information:

% time foo



Measuring Execution Time

Row Major (N=8192)

```
$ g++ row-major.cpp -o row-major -O3  
$ time ./row-major  
Total is: 16777216  
0.135u 0.140s 0:00.27 100.0%  
0+0k 0+0io 1pf+0w
```

Column Major (N=8192)

```
$ g++ col-major.cpp -o col-major -O3  
$ time ./col-major  
Total is: 16777216  
2.05u 0.137s 0:02.18 100.0%  
0+0k 0+0io 1pf+0w
```

Measuring Time in Code

Execution time can be measured inside code, usually by recording the beginning and end of a code segment and taking the difference.

```
#include <ctime>

int main() {
    clock_t start = clock();
    ...
    clock_t end = clock();
    cout << (end - start) / (double)CLOCKS_PER_SEC << endl;
}
```

```
$ ./row-major
Total is: 16777216
0.26442
```

Measuring Time in Code

Could also use `time_t` and `time`. Limited to the number of elapsed seconds.

```
#include <ctime>

int main() {
    time_t start, end;
    time(&start);
    ...
    time(&end)
    cout << difftime(start, end) << endl;
}
```

```
$ ./row-major
```

```
Total is: 16777216
```

```
0
```

Measuring Time in Code

`gettimeofday` sets a `timeval` which contains the current time in `tv_sec` (seconds) and `tv_usec` (microseconds).

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

```
int main() {  
    timeval start, end;  
    gettimeofday(&start);  
    ...  
    gettimeofday(&end)  
    cout << (end.tv_sec - start.tv_sec) +  
            (end.tv_usec - start.tv_usec)/1000000.0 << endl;  
}
```

```
$ ./row-major
```

```
Total is: 16777216
```

```
0.262752
```

Profiling

While manually timing code execution is often useful, there are a number of profiling tools that can give us more information if we are looking to optimise our programs

Helpful for identifying where our program is spending most of its time to make sure we are actually optimising the right part of the program.

GNU Prof

GNU provides a tool called gprof or the GNU profiler. This tool can provide information on how many times functions are called, how much of the run-time is spent in different functions etc.

This can be helpful for spotting bugs and analysing large programs without needing to resort to reading the source code or adding extra function calls to time different parts of the program.

GNU Prof

To make use of gprof, the code must be compiled and linked with profiling enabled. This can be done by simply adding the flags `-pg` to `gcc`. The program must be run to generate profile data.

```
$ g++ row-major.cpp -pg -o row-major
$ ./row-major
Total is: 16777216
0.661369
$ ls
gmon.out row-major row-major.cpp
```

GNU Prof

gprof can then be used to analyse this data.

First it gives a flat profile showing which functions take the most time and also a Call Graph.

```
$ gprof row-major -b
```

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
62.95	0.40	0.40	1	402.88	402.88	init(unsigned int*)
37.77	0.64	0.24	1	241.73	241.73	sum(unsigned int*)
0.00	0.64	0.00	1	0.00	0.00	__GLOBAL__sub_I__Z4initPj
0.00	0.64	0.00	1	0.00	0.00	__static_initialization_and_destruction_0(int, int)

GNU Prof

Call graph

granularity: each sample hit covers 2 byte(s) for 1.55% of 0.64 seconds

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.00	0.64		main [1]
		0.40	0.00	1/1	init(unsigned int*) [2]
		0.24	0.00	1/1	sum(unsigned int*) [3]

		0.40	0.00	1/1	main [1]
[2]	62.5	0.40	0.00	1	init(unsigned int*) [2]

		0.24	0.00	1/1	main [1]
[3]	37.5	0.24	0.00	1	sum(unsigned int*) [3]

		0.00	0.00	1/1	__libc_csu_init [16]
[8]	0.0	0.00	0.00	1	_GLOBAL__sub_I__Z4initPj [8]
		0.00	0.00	1/1	
					__static_initialization_and_destruction_0(int, int) [9]

		0.00	0.00	1/1	_GLOBAL__sub_I__Z4initPj [8]
[9]	0.0	0.00	0.00	1	
					__static_initialization_and_destruction_0(int, int) [9]

Valgrind

Valgrind is a tool that looks through code for memory errors or memory leaks. However, it also provides two other tools, callgrind and cachegrind

These tools can be used to analyse the function calls in your program and also simulate the number of L1/L2 caches and count the cache hits/misses.

Valgrind

Rather than simply analysing the amount of time spent executing certain blocks of code, we can get some insight as to why that code is slow.

Callgrind can simulate the cache give you information about cache hits/misses and can produce annotated source code to show which lines are causing problems.

Valgrind

```
$ valgrind --tool=callgrind --simulate-cache=yes ./row-major
```

```
Total is: 16777216
```

```
Events      : Ir Dr Dw I1mr D1mr D1mw I1Lmr DLmr DLmw
```

```
Collected : 1813569768 738575009 268589249 1469 4204140 4196100 1451  
4200593 4195665
```

```
I   refs:      1,813,569,768
```

```
I1  misses:      1,469
```

```
LLi misses:      1,451
```

```
I1  miss rate:      0.0%
```

```
LLi miss rate:      0.0%
```

```
D   refs:      1,007,164,258 (738,575,009 rd + 268,589,249 wr)
```

```
D1  misses:      8,400,240 ( 4,204,140 rd + 4,196,100 wr)
```

```
LLd misses:      8,396,258 ( 4,200,593 rd + 4,195,665 wr)
```

```
D1  miss rate:      0.8% ( 0.5% + 1.5% )
```

```
LLd miss rate:      0.8% ( 0.5% + 1.5% )
```

```
LL  refs:      8,401,709 ( 4,205,609 rd + 4,196,100 wr)
```

```
LL  misses:      8,397,709 ( 4,202,044 rd + 4,195,665 wr)
```

```
LL  miss rate:      0.2% ( 0.1% + 1.5% )
```

Valgrind

This doesn't give us too much useful information, only overall statistics. However, we can get callgrind to analyse our program and give us line by line statistics.

```
$ g++ row-major.cpp -o row-major -g
$ valgrind --tool=callgrind --simulate-cache=yes ./row-major
==28115== Callgrind, a call-graph generating cache profiler
...
$ callgrind_annotate --auto=yes callgrind.out.28115
```

Valgrind

This will output the cache statistics for each line of our program:

Ir	I cache reads (instructions executed)
I1mr	I1 cache read misses (instruction wasn't in I1 cache but was in L2)
I2mr	L2 cache instruction read misses (instruction wasn't in I1 or L2 cache, had to be fetched from memory)
Dr	D cache reads (memory reads)
D1mr	D1 cache read misses (data location not in D1 cache, but in L2)
D2mr	L2 cache data read misses (location not in D1 or L2)
Dw	D cache writes (memory writes)
D1mw	D1 cache write misses (location not in D1 cache, but in L2)
D2mw	L2 cache data write misses (location not in D1 or L2)

Valgrind

row-major

-- Auto-annotated source: row-major.cpp

Ir	Dr	Dw	Ilmr	Dlmr	Dlmw	ILmr	DLmr	DLmw	
.	#include <iostream>
.	
.	const int N = 8192;
.	
3	0	2	void init(unsigned int*p) {
40,966	8,193	8,193	1	0	0	1	.	.	for(int y = 0; y < N; y++) {
335,593,472	67,117,056	67,117,056	for(int x = 0; x < N; x++) {
603,979,776	335,544,320	67,108,864	0	0	4,194,304	0	0	4,194,304	p[y*N + x] = x*y;
.	}
.	}
2	2	0	0	1	0	0	1	.	}
.	
3	0	2	unsigned int sum(unsigned int *p) {
1	0	1	unsigned int total = 0;
40,966	8,193	8,193	1	0	0	1	.	.	for(int y = 0; y < N; y++) {
335,593,472	67,117,056	67,117,056	for(int x = 0; x < N; x++) {
536,870,912	268,435,456	67,108,864	0	4,194,305	0	0	4,194,305	.	total += p[y*N + x];
.	}
.	}
1	1	return total;
2	2	0	0	1	0	0	1	.	}

Valgrind

col-major

-- Auto-annotated source: col-major.cpp

Ir	Dr	Dw	ILmr	Dlmr	Dlmw	ILmr	DLmr	DLmw
. #include <iostream>
.
. const int N = 8192;
.
3	0	2 void init(unsigned int*p) {
40,966	8,193	8,193	1	0	0	1	.	. for(int x = 0; x < N; x++) {
335,593,472	67,117,056	67,117,056 for(int y = 0; y < N; y++) {
603,979,776	335,544,320	67,108,864	0	0	67,108,863	0	0	67,108,863 p[y*N + x] = x*y;
. }
. }
2	2	0	0	1	0	0	1	. }
.
3	0	2 unsigned int sum(unsigned int *p) {
1	0	1 unsigned int total = 0;
40,966	8,193	8,193	1	0	0	1	.	. for(int x = 0; x < N; x++) {
335,593,472	67,117,056	67,117,056 for(int y = 0; y < N; y++) {
536,870,912	268,435,456	67,108,864	0	67,108,864	0	0	67,108,864	. total += p[y*N + x];
. }
. }
1	1 return total;
2	2	0	0	1	0	0	1	. }

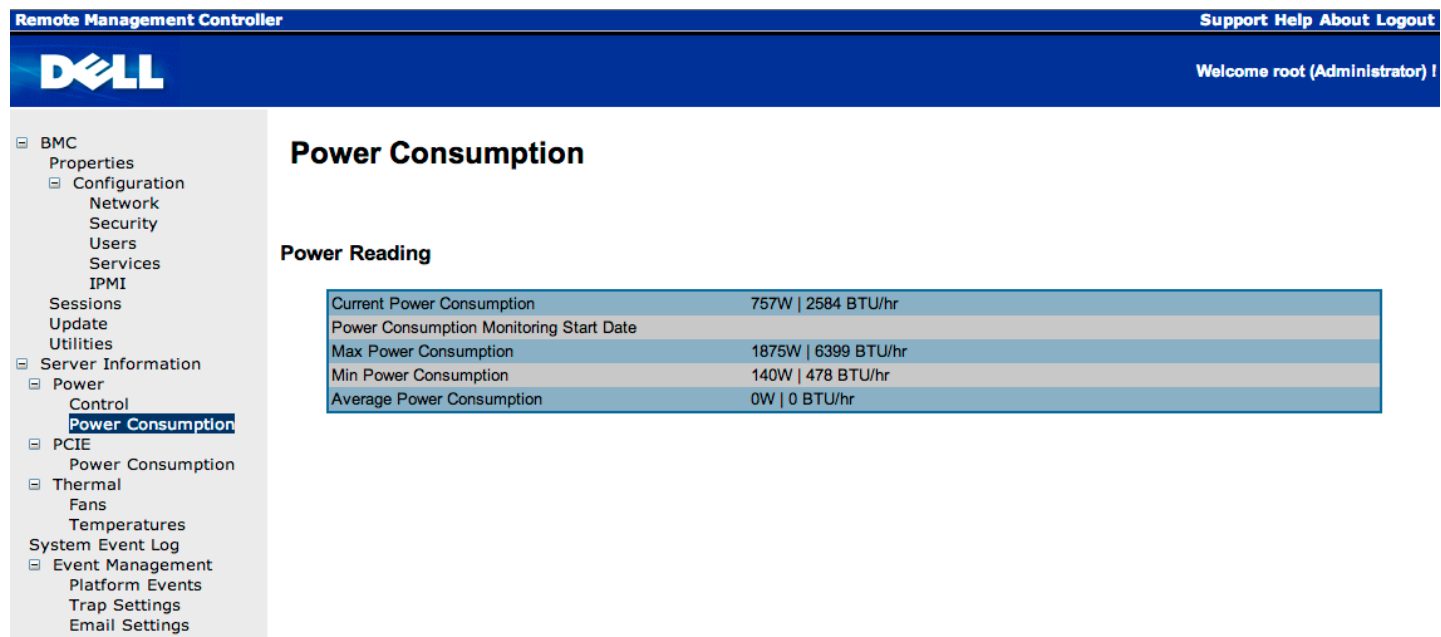
Measuring Power Consumption

The power consumption for a program is somewhat harder to measure. Power consumption is of increasing concern for many computing facilities.

Unfortunately power consumption of individual programs is either not available, or can be hard to measure without significantly impacting performance.

Measuring Power Consumption

Instead the overall power consumption of the machine is often measured. Some machines (such as servers) may have power usage statistics available:



The screenshot displays the Dell Remote Management Controller (iDRAC) web interface. The top navigation bar includes the Dell logo, the text "Remote Management Controller", and links for "Support", "Help", "About", and "Logout". Below the navigation bar, the user is logged in as "root (Administrator)". The left sidebar contains a tree view of the interface's sections, with "Power Consumption" highlighted under the "Power" section. The main content area is titled "Power Consumption" and features a "Power Reading" table.

Power Reading	
Current Power Consumption	757W 2584 BTU/hr
Power Consumption Monitoring Start Date	
Max Power Consumption	1875W 6399 BTU/hr
Min Power Consumption	140W 478 BTU/hr
Average Power Consumption	0W 0 BTU/hr

Measuring Power Consumption

Some power supply units display the current power draw or additional units such as the Kill-A-Watt can be used.



Summary

Performance analysis is an important part of quantifying the performance of a program.

Making sure you use the right tools is vital for generating accurate and reliable performance data.

Profiling tools may not always give you new insights into your program but can be useful for confirming your suspicions about why your code is running slow.