Arrays and performance in C

Ying Pei Lin

Fall 2024

Clock Accuracy

Before measuring the time required to access random elements in an array, we need determine the accuracy of the clock. This can be completed by calling the `clock_gettime()` function two times and calculate the time difference between the two calls. The following code snippet is used to test the accuracy of the clock and the result is shown in Figure 1.

```
int main(int argc, char *argv[]) {
  for(int i = 0; i < 1000; i++) {
    clock_gettime(CLOCK_MONOTONIC, &t_start);
    clock_gettime(CLOCK_MONOTONIC, &t_stop);
    long wall = nano_seconds(&t_start, &t_stop);
    printf("%ld ns\n", wall);
  }
}
```

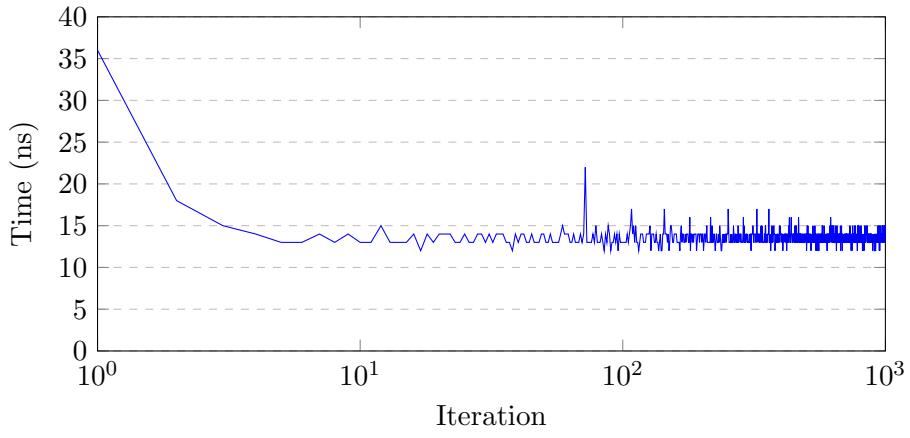

Figure 1: Time difference between two calls of clock_gettime()

The time difference between two calls of `clock_gettime()` is about 15 ns. The difference includes the time needed to execute the system call, read

the hardware clock and return the result. At the begining, The CPU will load the function into memory, if the function is not inside the cache (cache miss), then the CPU needs to get the function from the main memory, which takes more time. This could be why the first few iterations are slower than the rest.

Besides, when the CPU is executing multiple threads, it has to manage context switching between processes, distribute its computational resources across all active tasks, and handle other system overheads, which can cause delay. It will take longer to execute the function. Therefore, the workload of the CPU can also affect the time it takes to execute the function.

Figure 2 is the comparison between using only the terminal to run the program and running the program while web browsing and IDE is open. To avoid the influence from the CPU workload and ensure the accuracy of the time measurement, we should run the program in a quiet enviroment with minimal background processes.

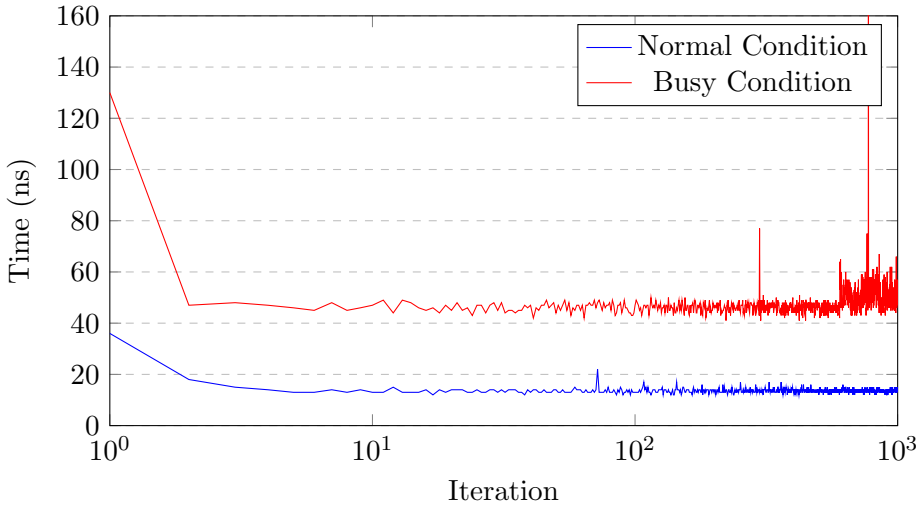

Figure 2: Comparison between running the program with and without CPU workload

Random Access

The following code snippet is used to measure the time required to access random elements in an array and the result is shown in Figure 4. To measure the time, we first create an array of size n with the value i at index i as the target we want to access. Then, we create another array of size *loop* with random values between 0 and $n - 1$ as the index array we use to access the target array. After that, we record the time before and after the loop that accesses the target array using the index array.

```
long bench(int n, int loop) {
  int *array = (int*)malloc(n*sizeof(int));
  for (int i = 0; i < n; i++) array[i] = i;

  int *indx = (int*)malloc(loop*sizeof(int));
  for (int i = 0; i < loop; i++) indx[i] = rand()%n;

  int sum = 0;
  clock_gettime(CLOCK_MONOTONIC, &t_start);
  for (int i = 0; i < loop; i++) sum += array[indx[i]];
  clock_gettime(CLOCK_MONOTONIC, &t_stop);

  if (sum == 0)
    return 0;

  long wall = nano_seconds(&t_start, &t_stop);
  return wall;
}

int main(int argc, char *argv[]) {
  int k = 10;
  int loop = 1000;
  for (int n = 1000; n <= 16384000; n *= 2) {
    long min_time = LONG_MAX;
    long max_time = 0;
    long total_time = 0;
    for (int i = 0; i < k; i++) {
      long wall = bench(n, loop);
      total_time += wall;
      if (wall < min_time) min_time = wall;
      if (wall > max_time) max_time = wall;
    }
    long avg_time = total_time/k;
    printf("%d %0.2f %0.2f %0.2f\n",
            n, (double)min_time/loop, (double)max_time/loop, (double)avg_time/loop
  }
}
```

The execution time is relatively stable around 6 ns for smaller arrays with size less than 16k elements. The time increases as the array size grows, when the size reach between 64k and 256k elements, the times ranged from 6 ns to 12 ns per operation, with some significant fluctuations at certain points. As we increase the array size to more than 512k elements, the time becomes more stable around 10 ns for each operation and when the size

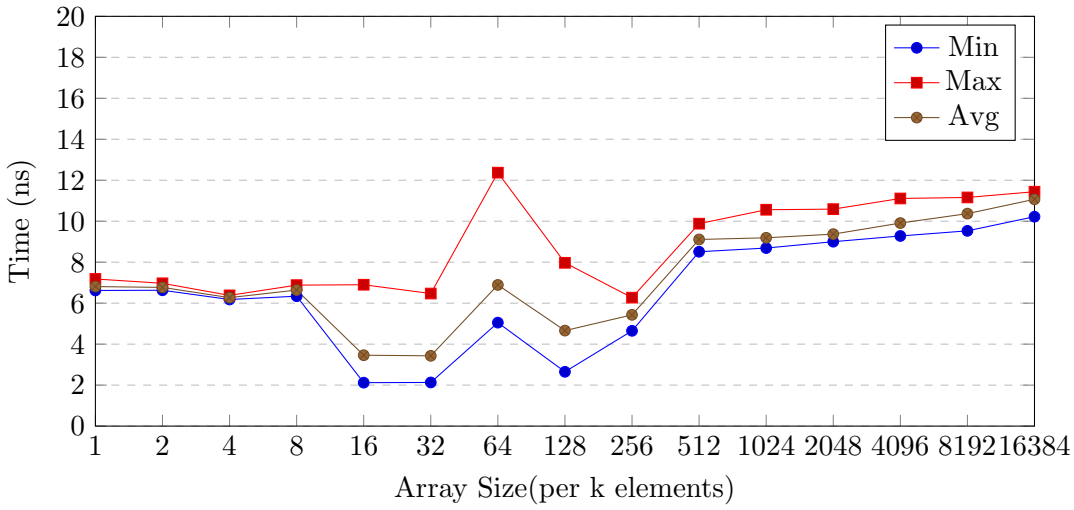

Figure 3: Random access time in an array

doubles, the time increase a little bit. If the time increases linearly while the array size doubles, the pattern is likely to be $O(log(n))$ and the time can be approximated by $T(n) = a + b \cdot log(n)$. We can do another experiment foucsing on the bigger array size to verify the pattern. The result is shown in Figure **??**.

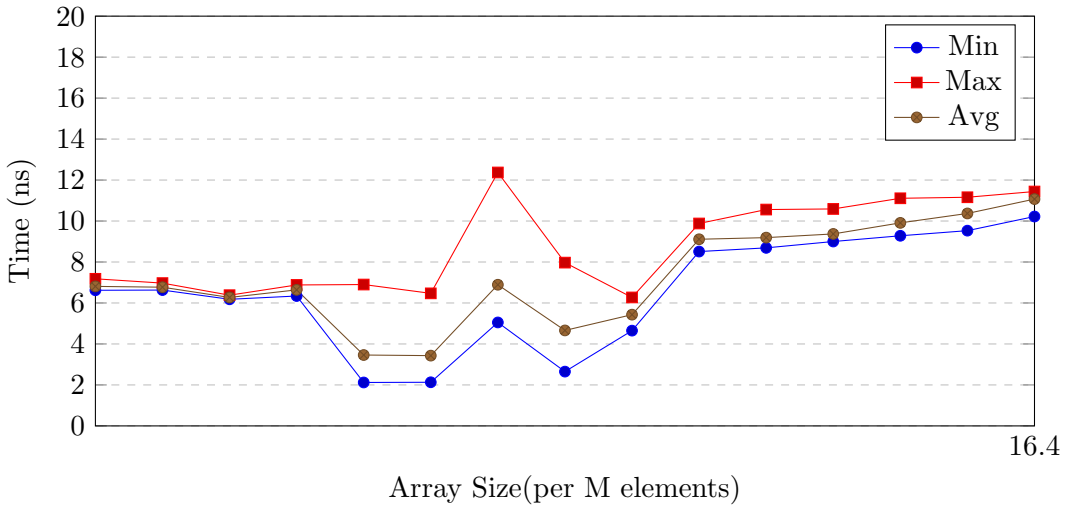

Figure 4: Random access time in an array

Search

Duplicates