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);
  }
}
```

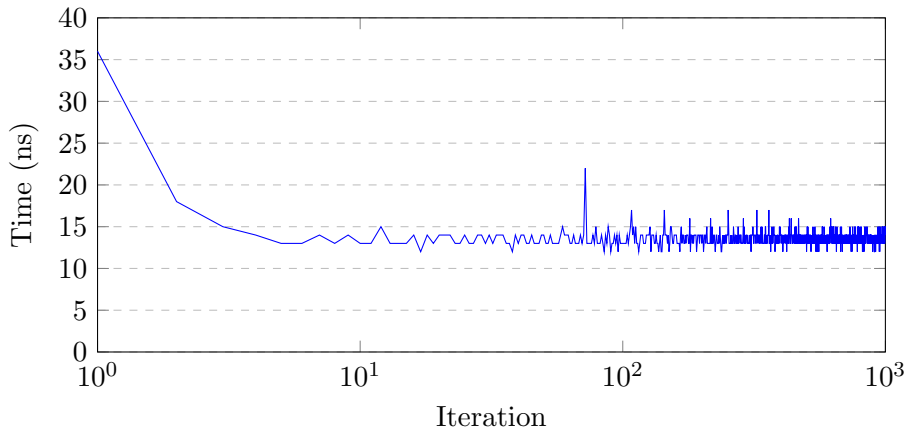

Figure 1: Time difference between two calls of clock_gettime()

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 beginning, 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.

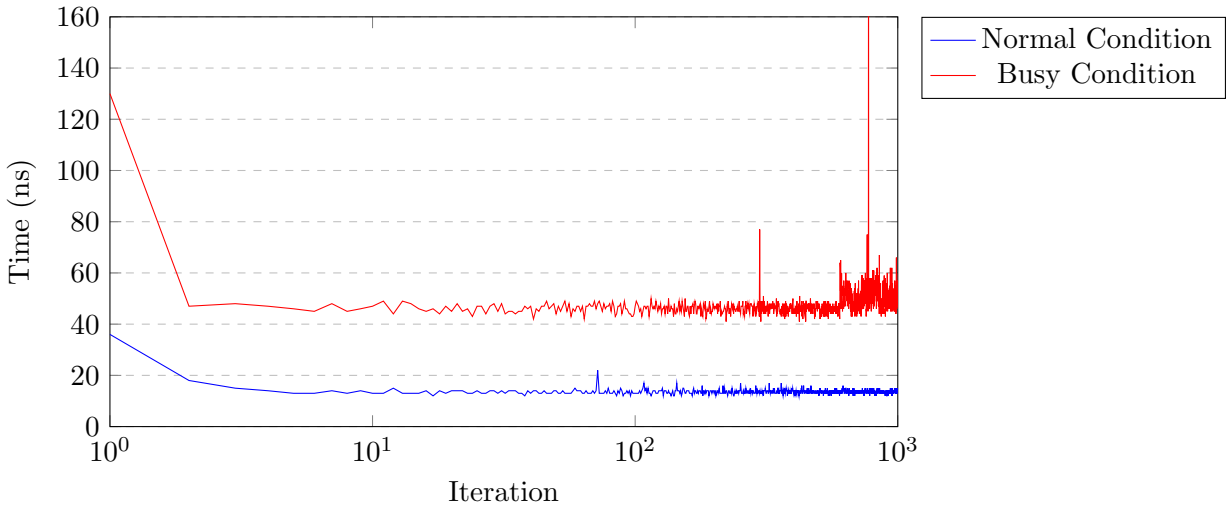

Figure 2: Comparison between running the program with and without CPU workload

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 3. 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)
  }
}
```

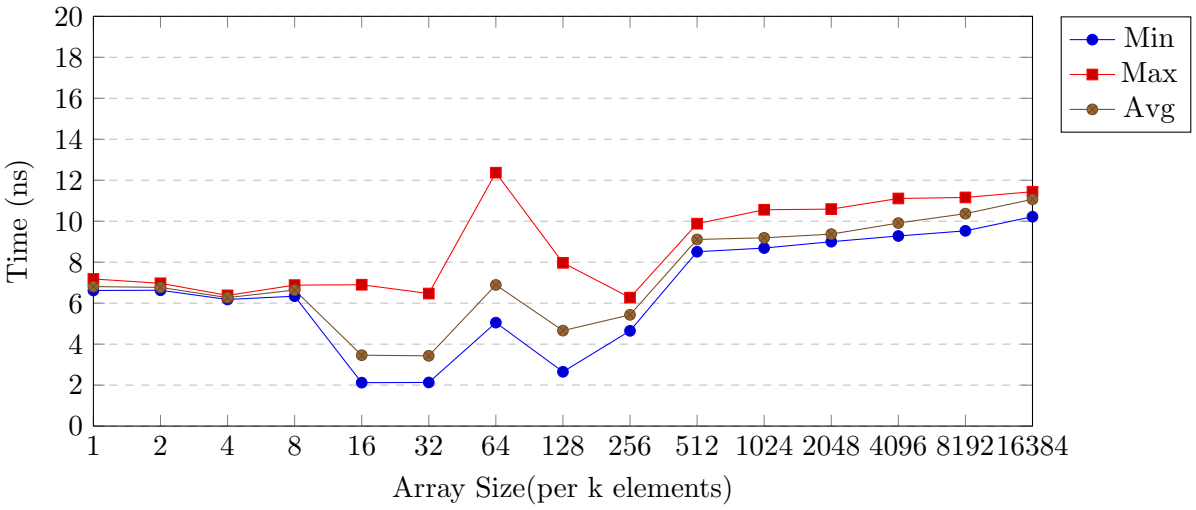

Figure 3: Random access time in an array

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 the array increases to more than 512k elements, the time becomes more stable around 10 ns for each operation and when the size doubles, the time increase a little bit. If the time increases linearly while the array size doubles, the pattern is likely to be $O(log(n))$, or more precisely, $O(log(n * loop))$.The time can be approximated by $T(n) = a + b \cdot log(n * loop)$.

We can do another experiment foucsing on the bigger array size to verify the pattern. The result is shown in Figure 4. I also ran the program with more bigger array size, however, my computer seems to be unable to handle it.

As the array size becomes larger, the minimum time increases more stable than the maximum time. Because there will be a best case that the operation is completed in the shortest time and no matter how good the other case is, they won't faster than the best case, but can be close to it. On the other hand, the worst case can be so bad that beyond our imagination, and the maximum time can be very large.

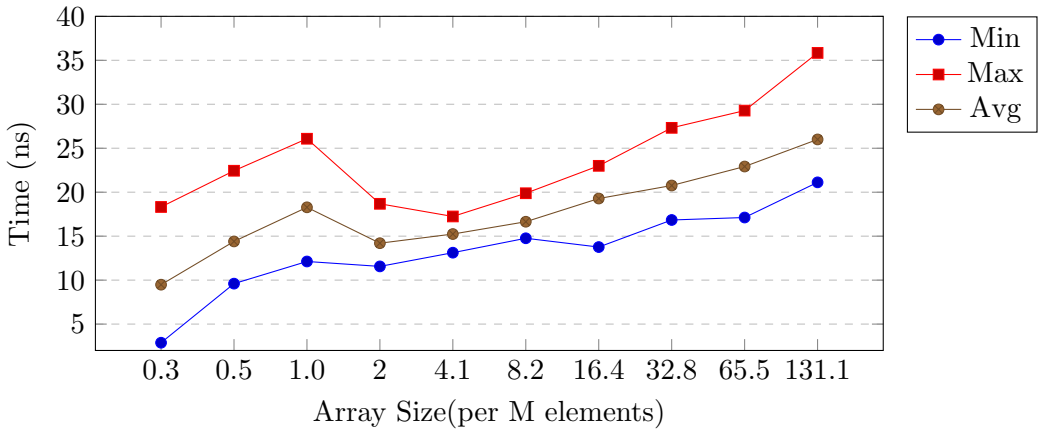

Figure 4: Random access time in an array with bigger size

Search for an item

The following code snippet is used to measure the time required to search for an item in an array, and the result is shown in Figure 5.

```
long search(int n, int loop) {
  int *array = (int*)malloc(n*sizeof(int));
  for (int i = 0; i < n; i++) array[i] = rand()%(n*2);

  int *keys = (int*)malloc(loop*sizeof(int));
  for (int i = 0; i < loop; i++) keys[i] = rand()%(n*2);

  int sum = 0;
  clock_gettime(CLOCK_MONOTONIC, &t_start);
  for (int i = 0; i < loop; i++) {
    int key = keys[i];
    for (int j = 0; j < n; j++) {
      if (key == array[j]) {
        sum++;
        break;
      }
    }
  }
  clock_gettime(CLOCK_MONOTONIC, &t_stop);
  if(sum == 0) return 0;
  long wall = nano_seconds(&t_start, &t_stop);
  return wall;
}
```

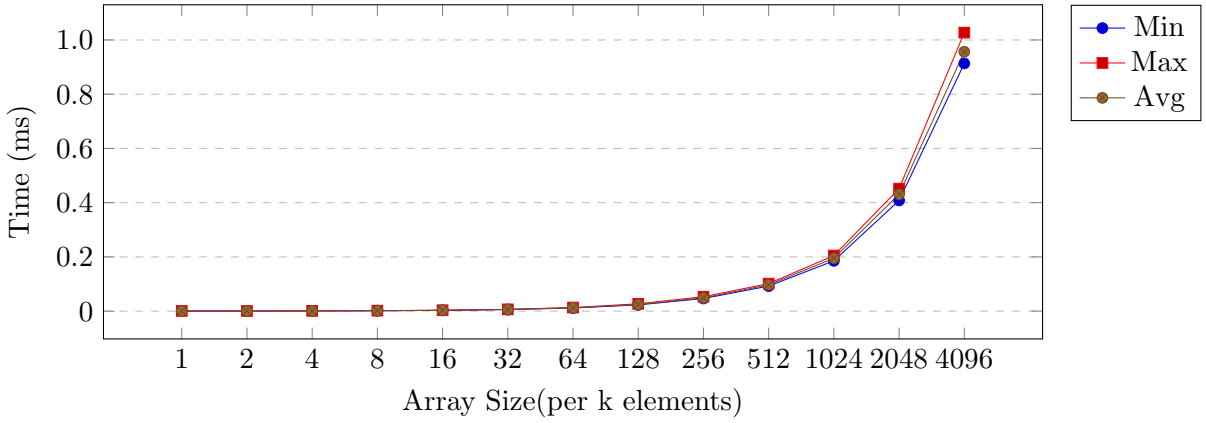

Figure 5: Search time in an array

The result is quite different from the random access time. When the size of the array doubles, the time required also doubles, which indicates that the time might be linearly related to the size of the array. To verify this, I change the size of array in each iteration (from double in each iteration to increase by a constant value) and the new result is shown in Figure 6.

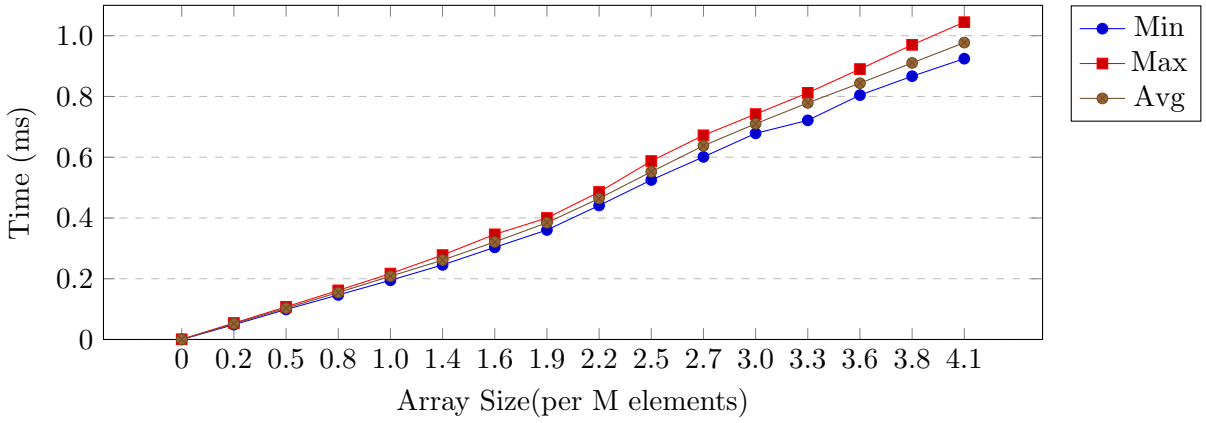

Figure 6: Search time in an array increasing by a constant value

The time required to search for an item in an array is linearly related to the size of the array when the size of the array is less than 4.1M elements according to the result. Therefore, the pattern is likely to be $O(n)$ and the time can be approximated by $T(n) = a + b \cdot n$.

As we searching from the start of the array to the end, the time required will be linearly related to where the target is located in the array. Assume there are n positions in the array and the probility of finding the target at each position is $1/n$, which is the same for all positions. Then, the expected position to find the target is

$$E[X] = \sum_{i=1}^{n} i \cdot \frac{1}{n} = \frac{1}{n} \sum_{i=1}^{n} i = \frac{n+1}{2} \tag{1}$$

Thus, the expected time to find the target $T(n)$ will be proportional to $\frac{n+1}{2}$, which is $O(n)$, or more precisely, $O(n * loop)$.

Search for duplicates

The following code snippet is used to measure the time required to search for duplicates in an array, and the result is shown in Figure 7.

Compared to the pervious search, the time required to search for duplicates in an array is much longer, due to the double nested loop and the time complexity is likely to be $O(n^2)$, or more precisely, $O(n^2 * loop)$. The time can be approximated by $T(n) = a + b \cdot n^2$.

```
long duplicates(int n, int loop) {
  int *array_a = (int*)malloc(n*sizeof(int));
  for (int i = 0; i < n; i++) array_a[i] = rand()%(n*2);

  int *array_b = (int*)malloc(n*sizeof(int));
  for (int i = 0; i < loop; i++) array_b[i] = rand()%(n*2);

  int sum = 0;
  clock_gettime(CLOCK_MONOTONIC, &t_start);
  for (int k = 0; k < loop; k++) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
      int key = array_a[i];
      for (int j = 0; j < n; j++) {
        if (key == array_b[j]) {
          sum++;
          break;
        }
      }
    }
  }
  clock_gettime(CLOCK_MONOTONIC, &t_stop);
  if(sum == 0) return 0;
  long wall = nano_seconds(&t_start, &t_stop);
  return wall;
}
```

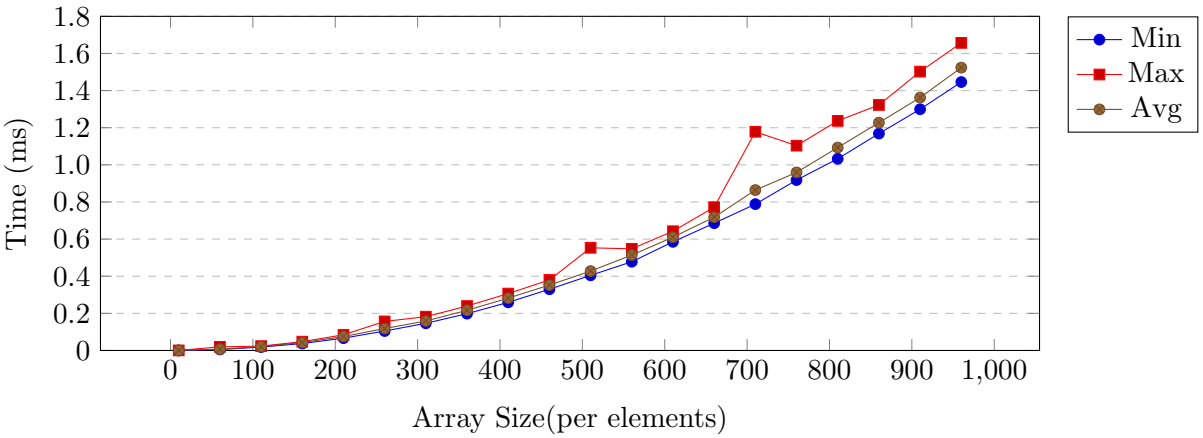

Figure 7: Search time for duplicates in an array