Metody Programowania Równoległego
Raport II - OpenMP

Maciej Trątnowiecki

AGH, Semestr Letni, 2022

1 Implementacja równoległego generowania losowych danych wejściowych

1.1 Definicja problemu

W ramach zadania przygotowaliśmy implementację programu generującego równolegle losowe dane wejściowe dla algorytmu sortującego. Implementacja przygotowana została w języku C, za wykorzystaniem interfejsu *OpenMP*. Kod implementacji dla poszczególnych wersji programu załączono w ostatniej sekcji sprawozdania. Tak przygotowany program wykorzystaliśmy do pomiaru czasu wykonania i przyśpieszenia algorytmu, dla różnych ustawień klauzuli *schedule*.

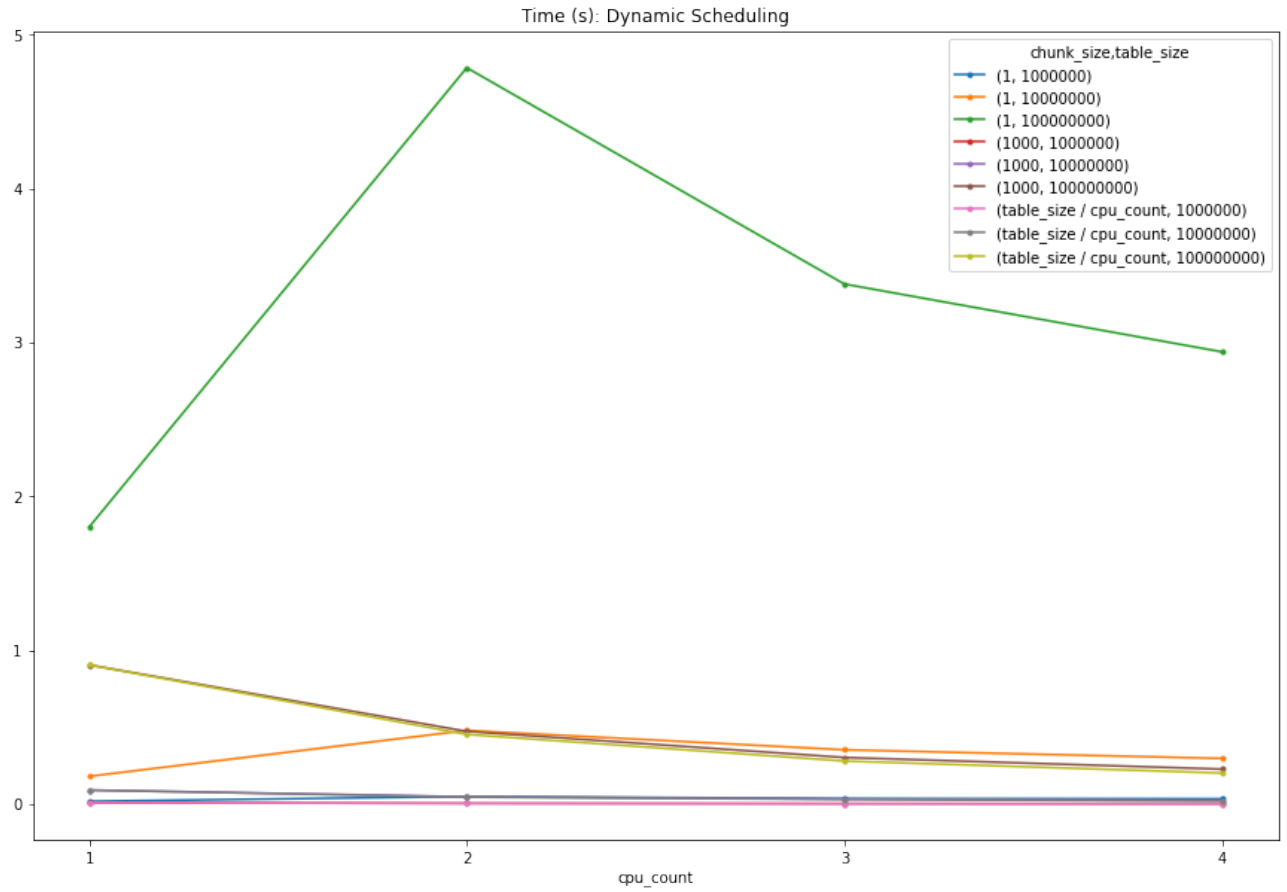
1.2 Pomiary i wyniki

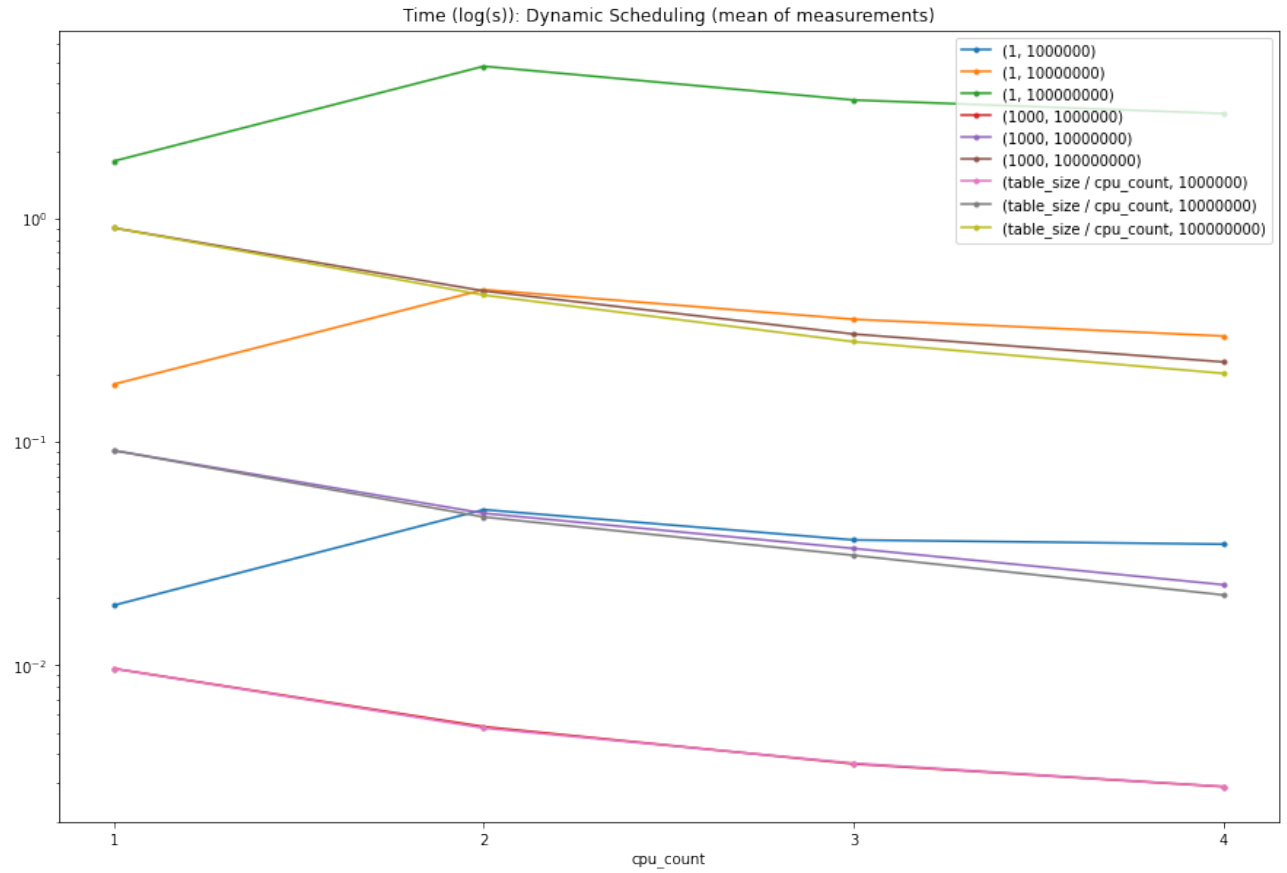
Każdy z pomiarów powtórzyliśmy dziesięciokrotnie dla danych ustawień parametrów programu. Program testowaliśmy dla ustawień klauzuli *schedule* ze zbioru *guided*, *dynamic* i *static*, oraz wartości parametru *chunk* z zakresu opisanego w tabeli poniżej.

Schedule	Chunk
Dynamic	1
Dynamic	1000
Dynamic	Rozmiar Problemu / Ilość CPU
Guided	1
Static	Rozmiar Problemu / Ilość CPU

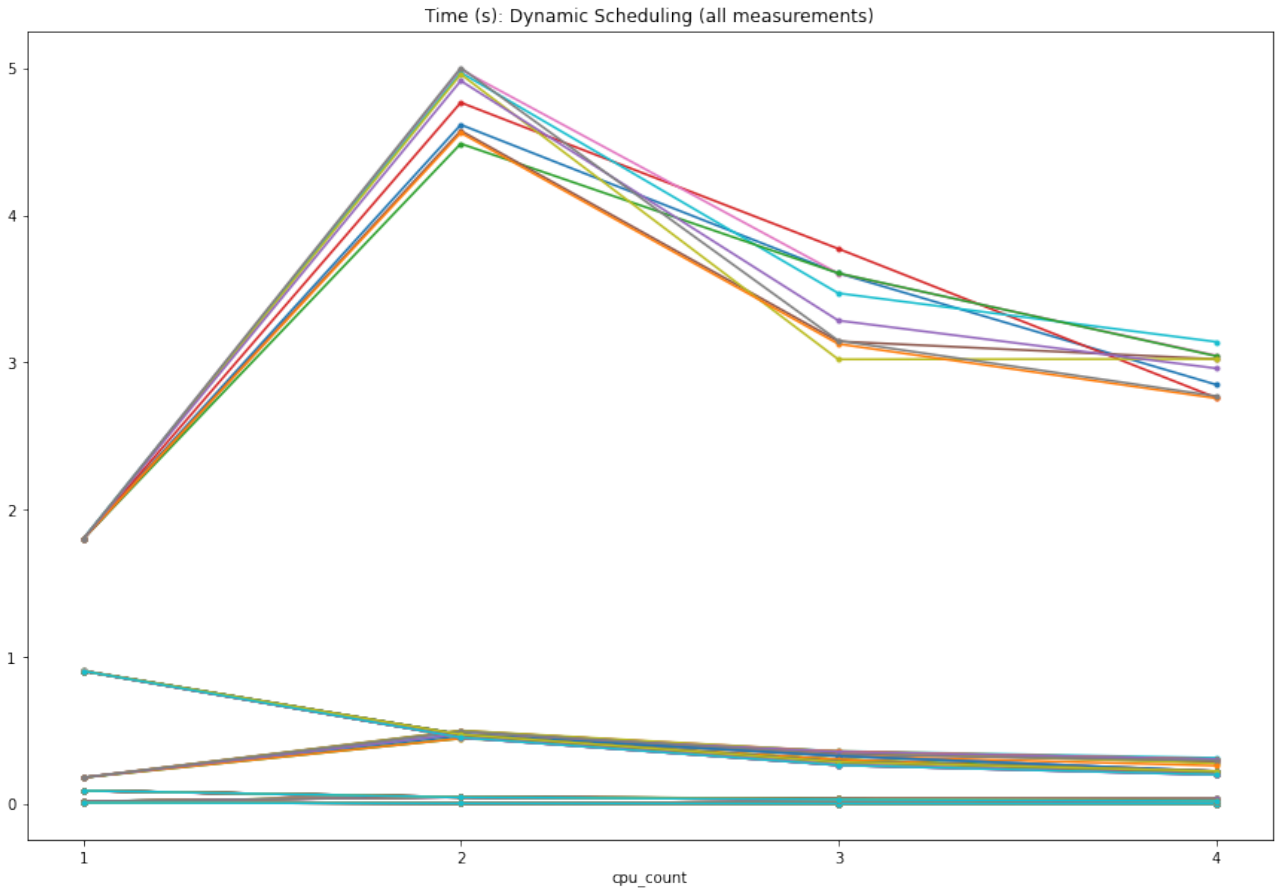
Tabela 1: Parametry programu wykorzystane do przygotowania pomiarów.

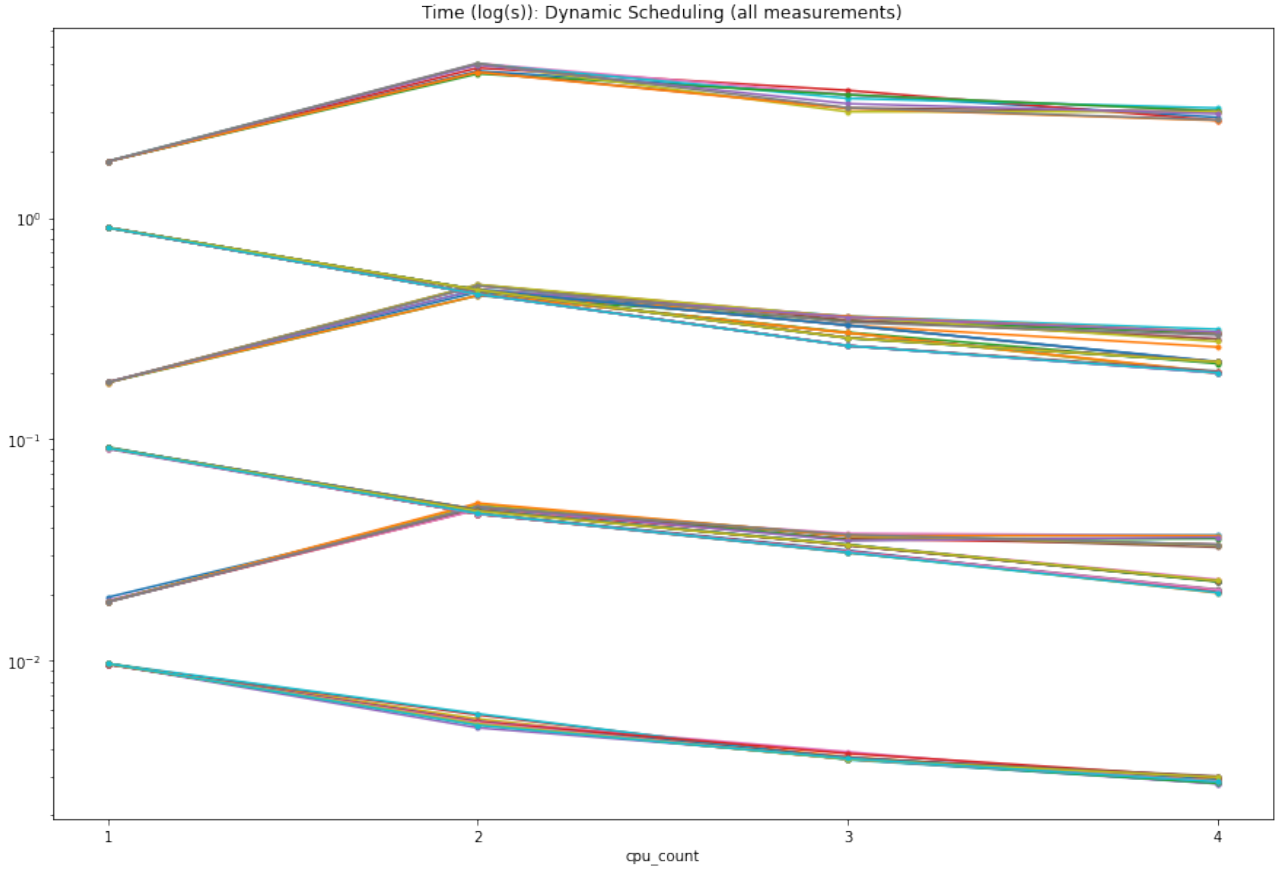
Program uruchamialiśmy dla problemów o rozmiarze 10^6, 10^7 i 10^8. Zebrane wyniki pomiarów czasu wykonania programu przedstawione zostały na poniższych wykresach.



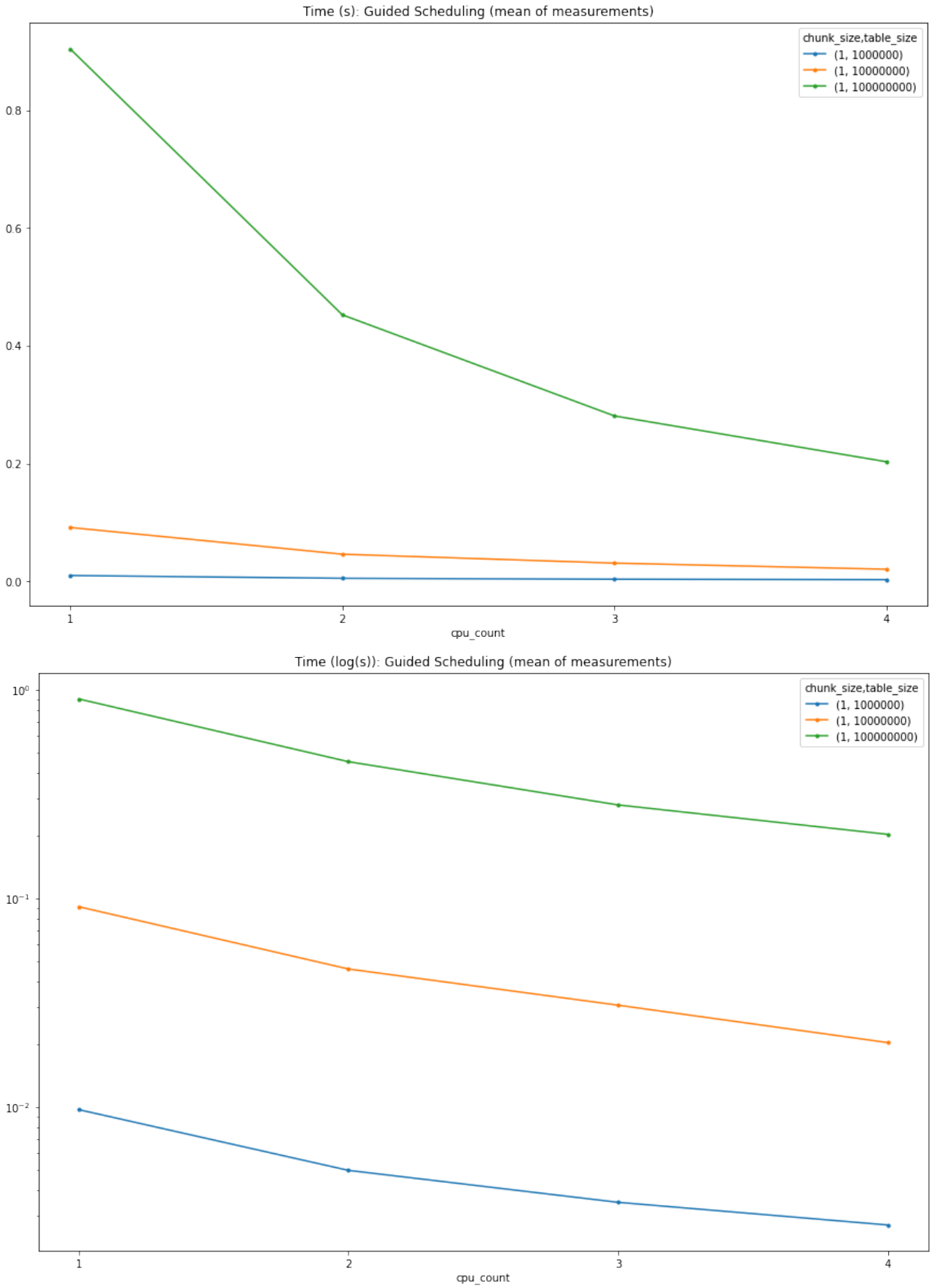


Rysunek 1: Pomiar czasu wykonania programu, Dynamic schedule, w zależności od parametrów. Uśrednione wartości z powtórzeń pomiarów.

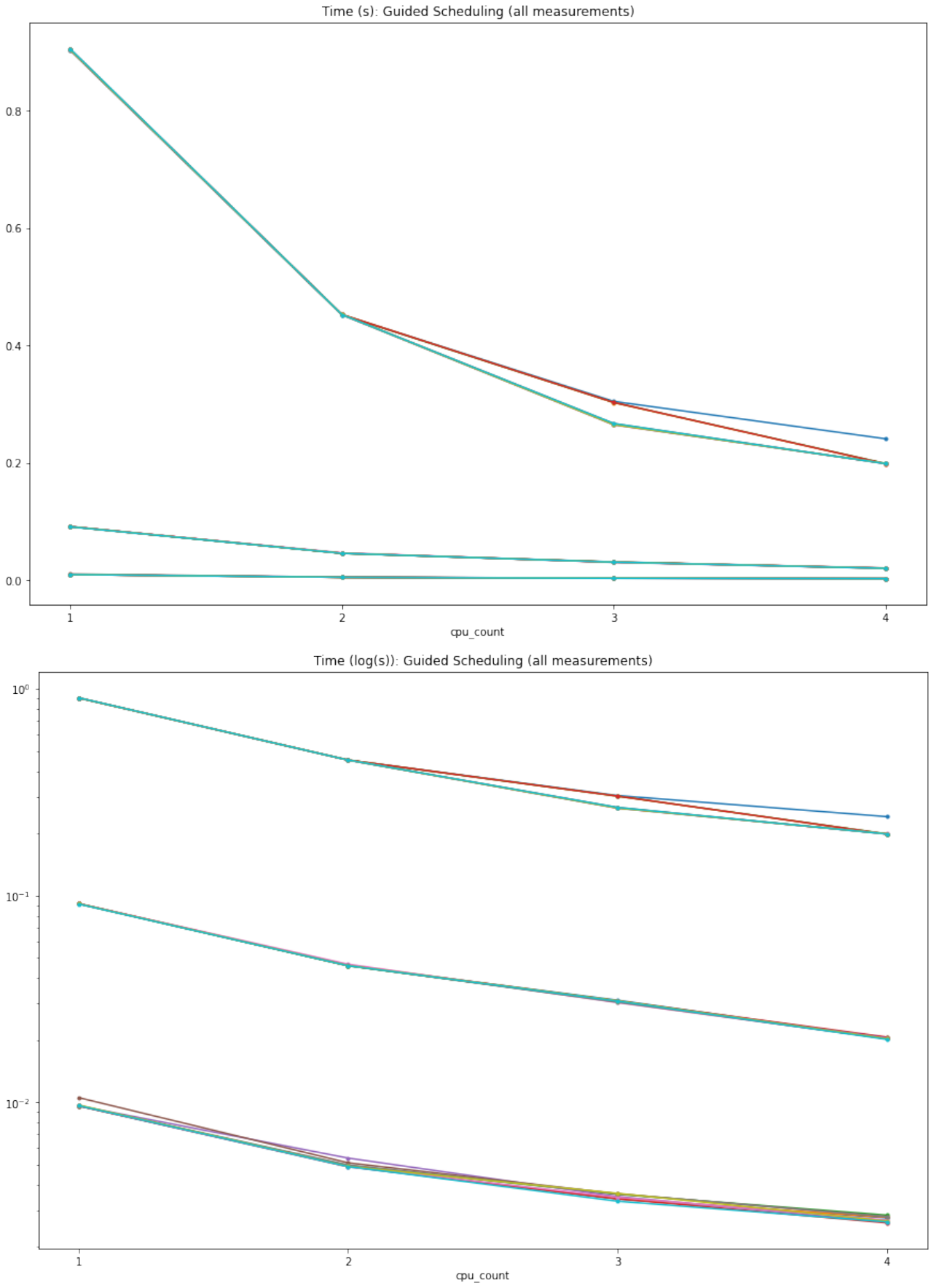




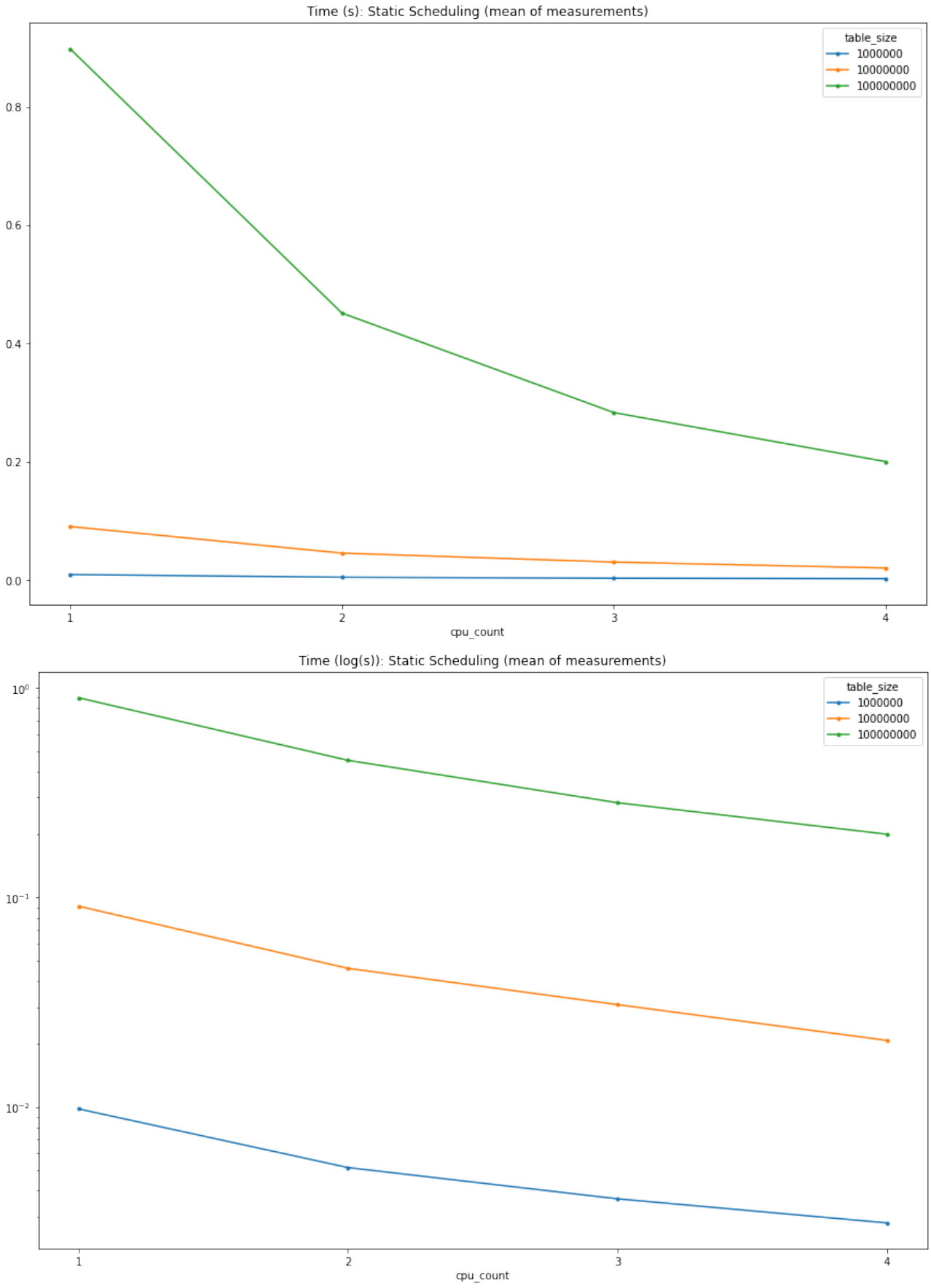
Rysunek 2: Pomiar czasu wykonania programu, Dynamic schedule. Wszystkie pomiary.



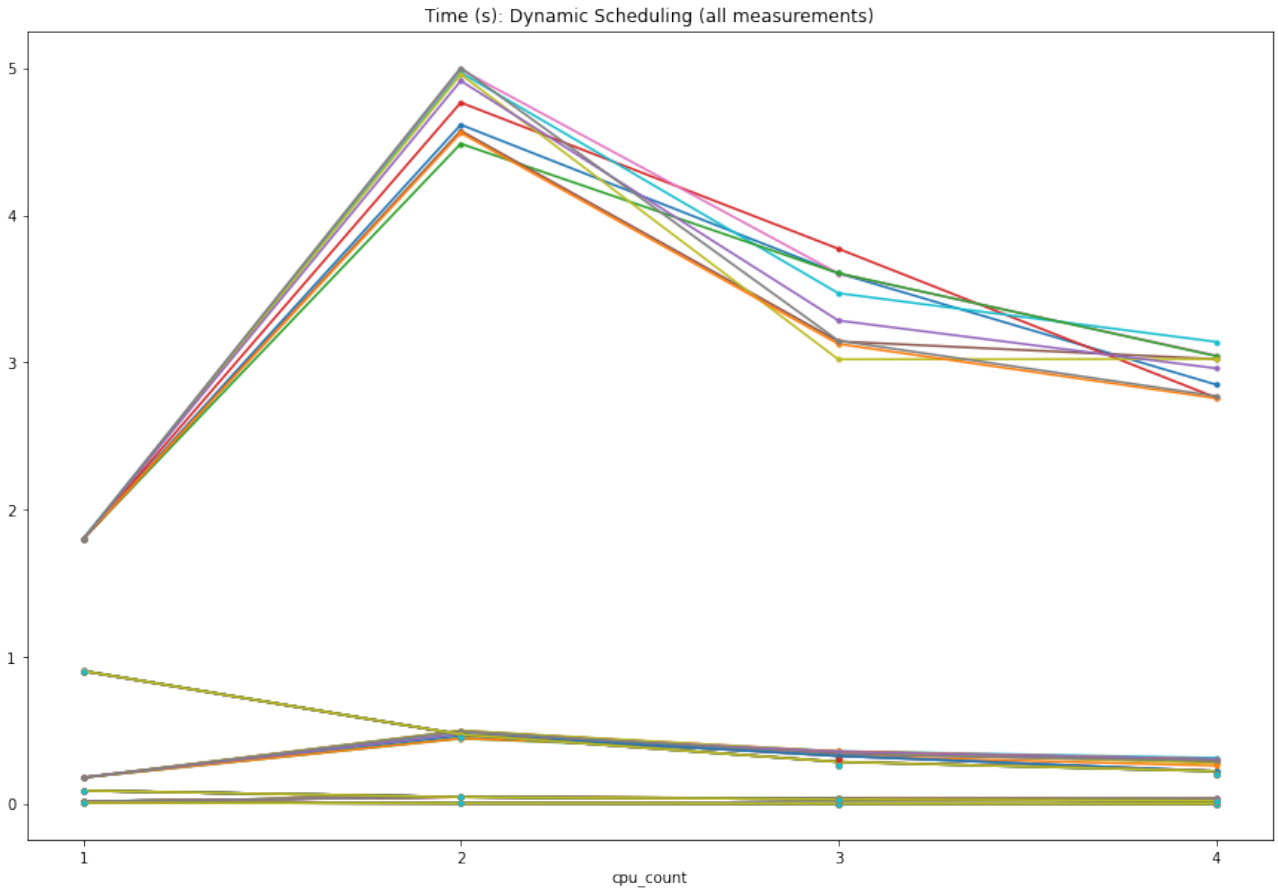
Rysunek 3: Pomiar czasu wykonania programu, guided schedule, w zależności od parametrów. Uśrednione wartości z powtórzeń pomiarów.

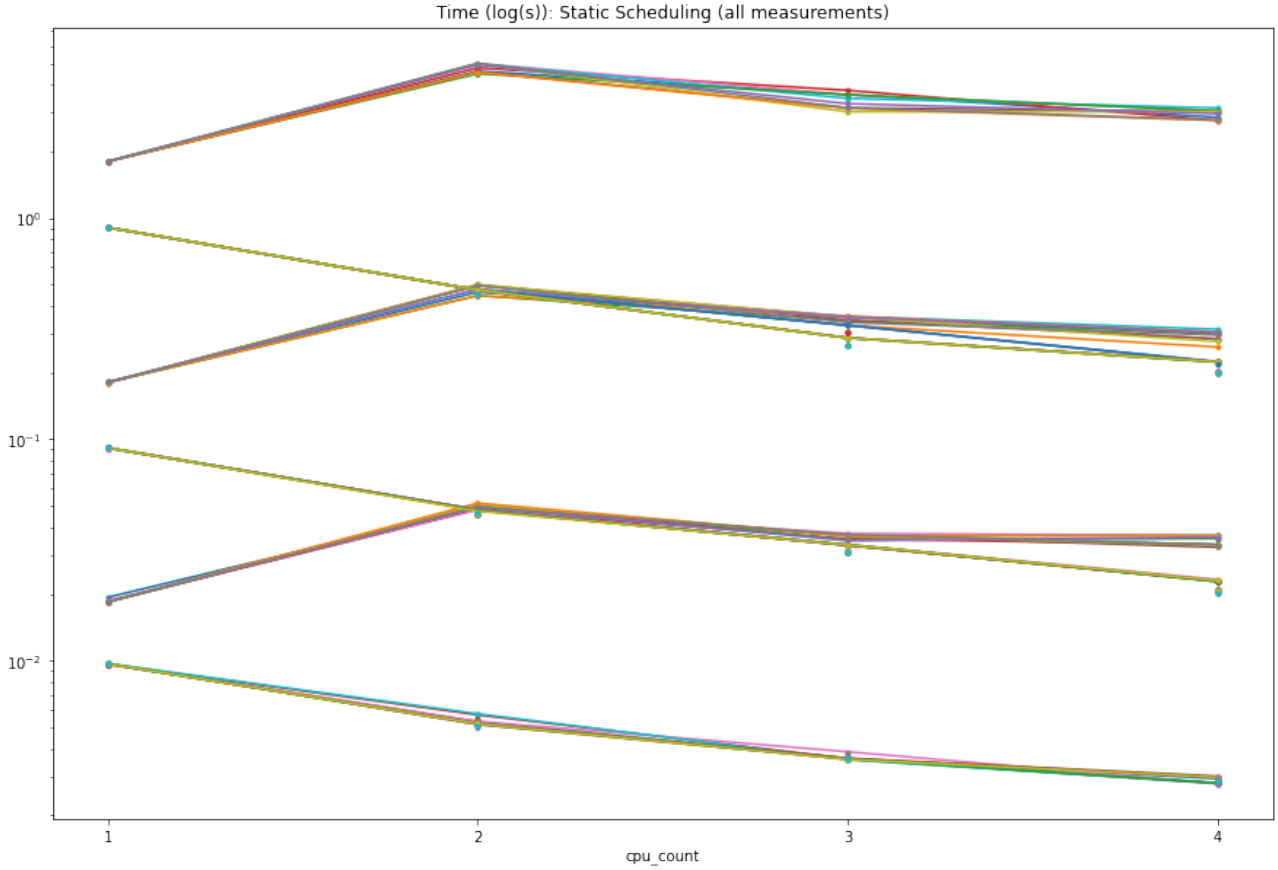


Rysunek 4: Pomiar czasu wykonania programu, guided schedule. Wszystkie pomiary.

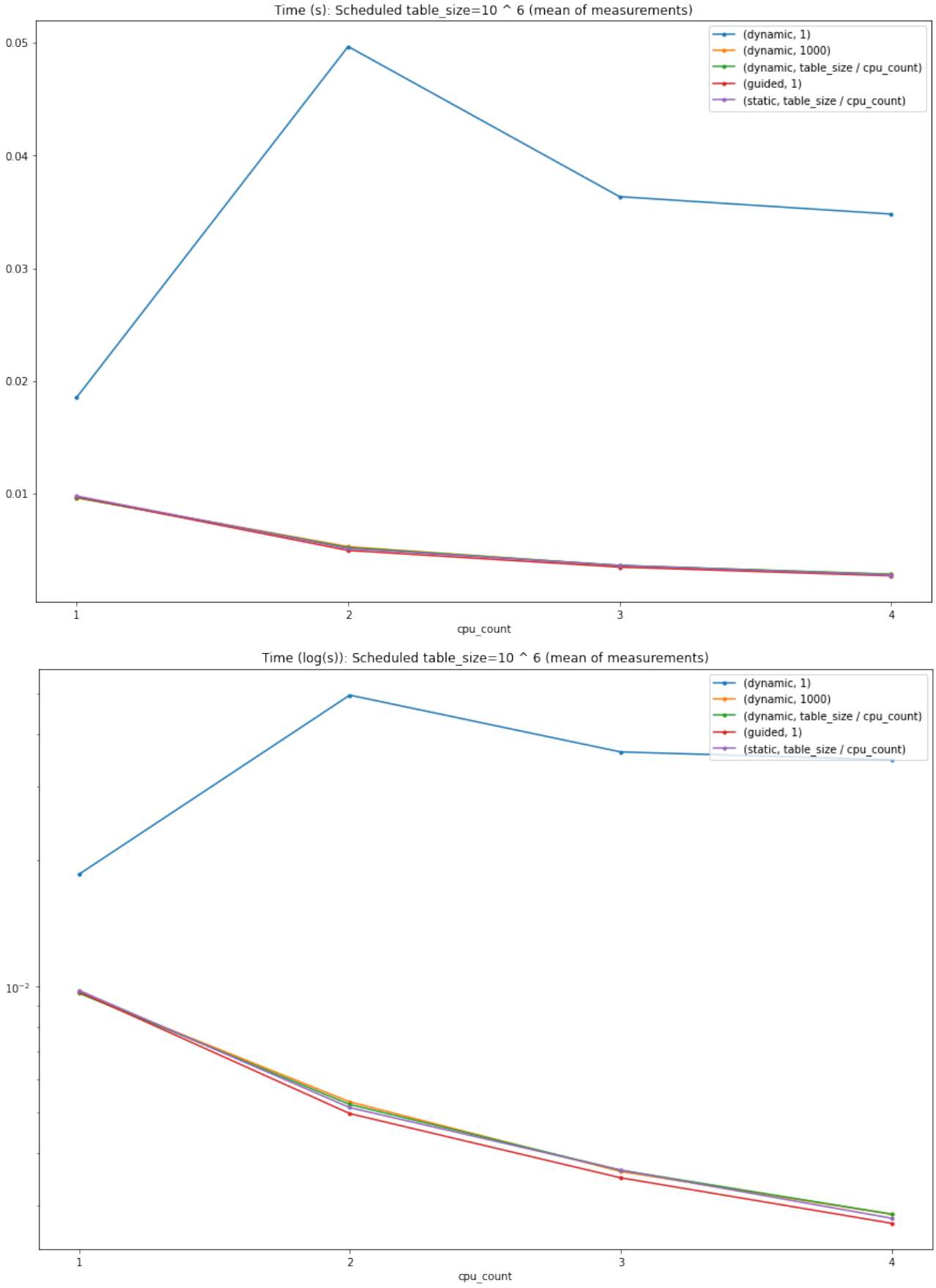


Rysunek 5: Pomiar czasu wykonania programu, static schedule, w zależności od parametrów. Uśrednione wartości z powtórzeń pomiarów.

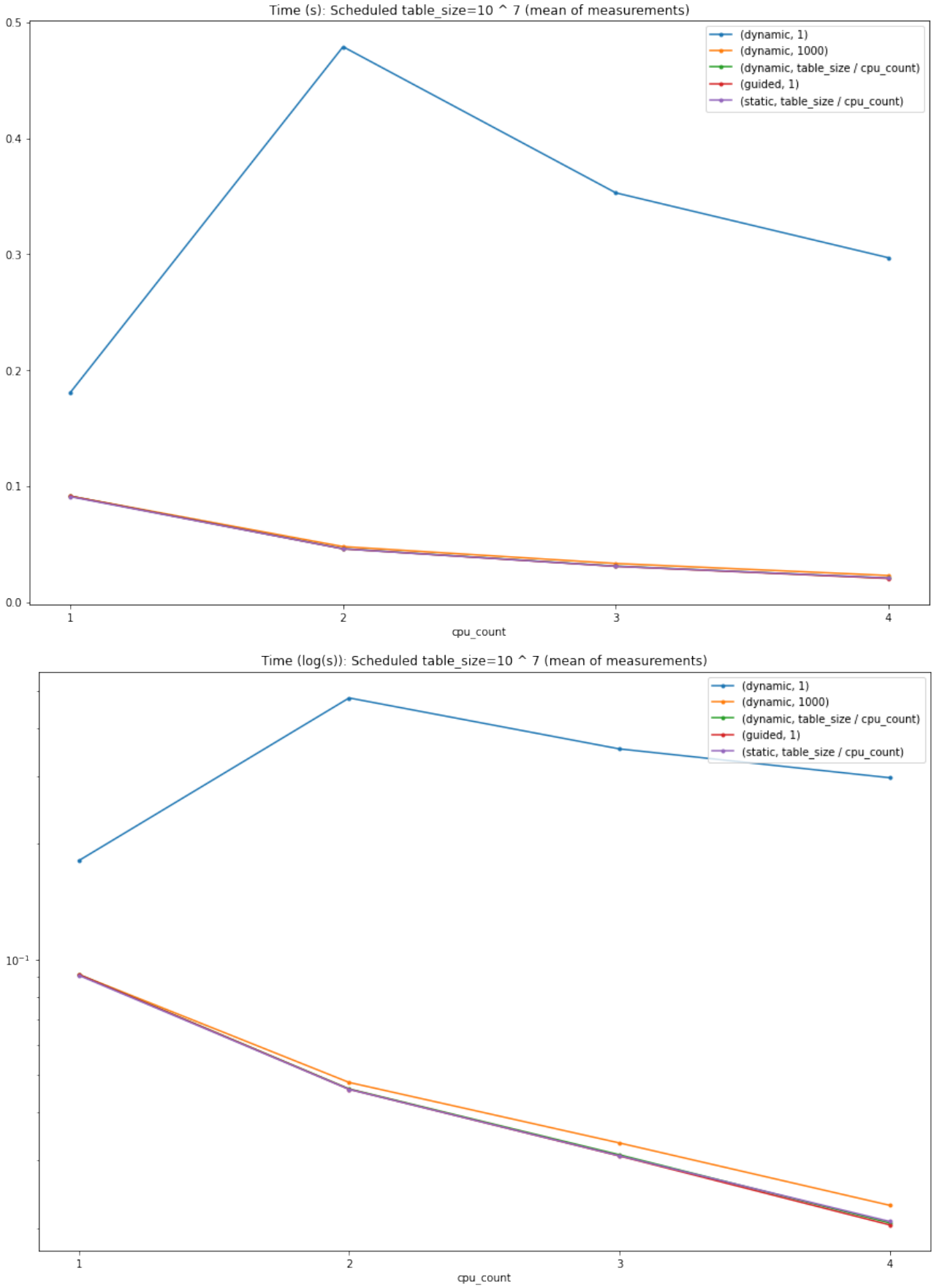




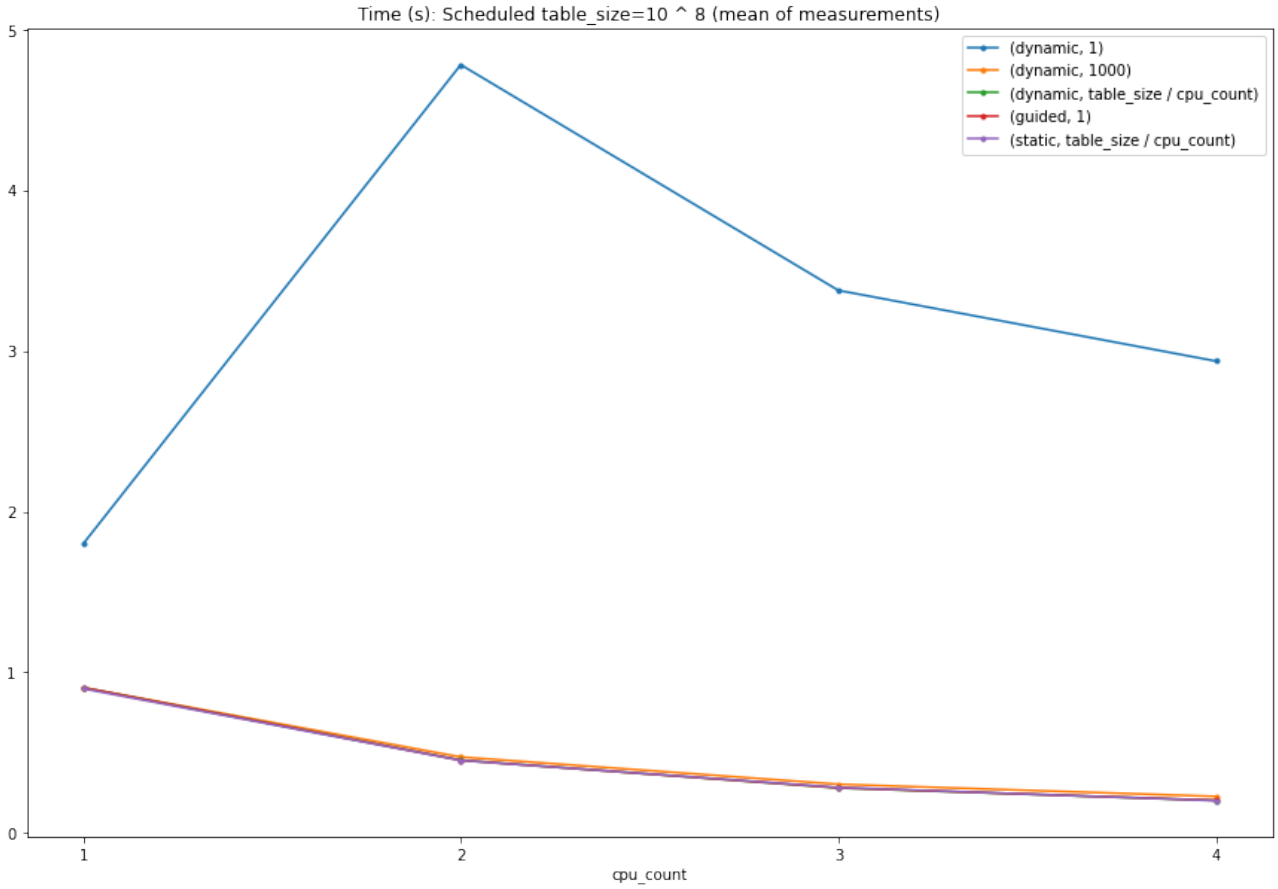
Rysunek 6: Pomiar czasu wykonania programu, static schedule. Wszystkie pomiary.

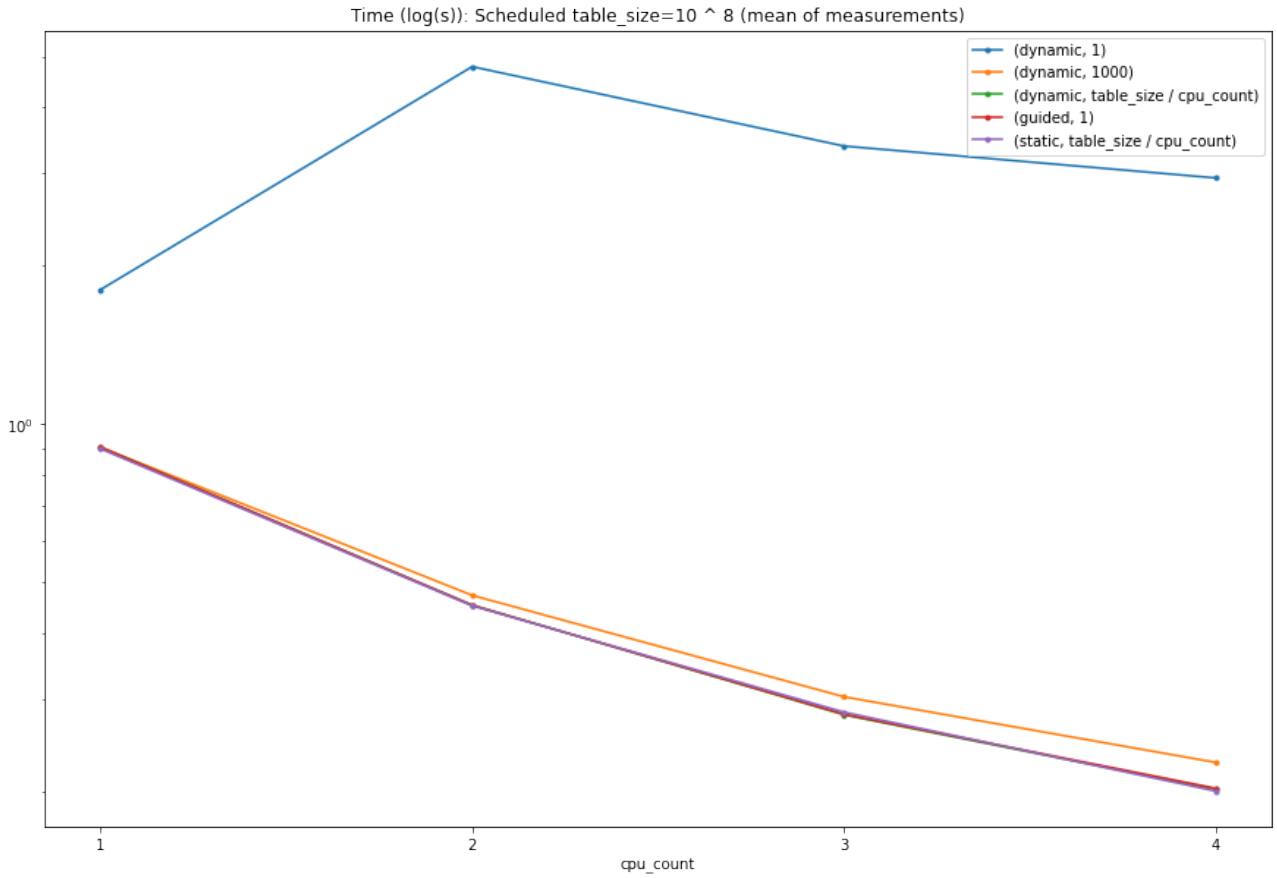


Rysunek 7: Pomiar czasu wykonania programu, dla problemu rozmiaru 10 ** 6, w zależności od parametrów. Uśrednione wartości z powtórzeń pomiarów.



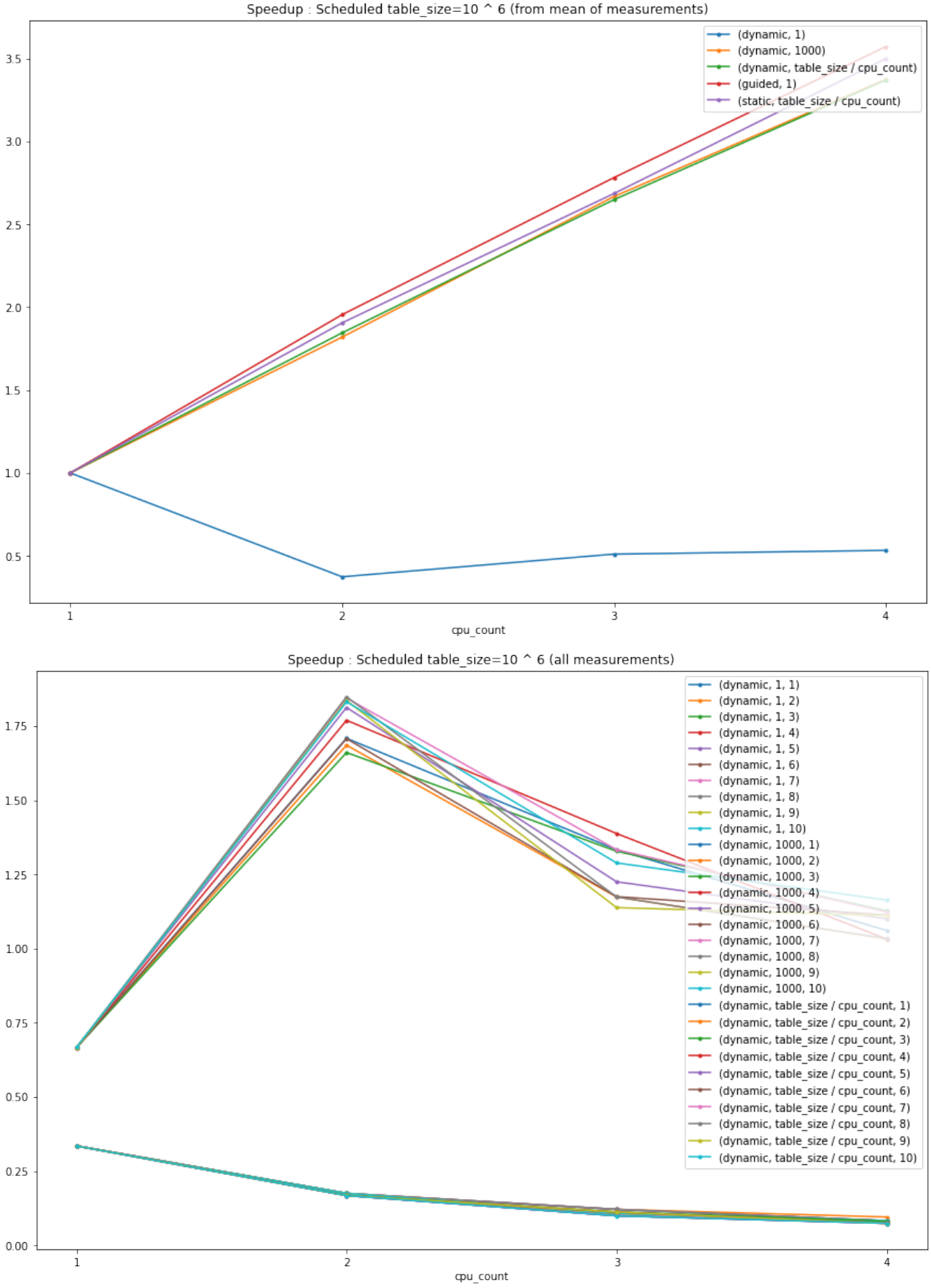
Rysunek 8: Pomiar czasu wykonania programu, dla problemu rozmiaru 10 ** 7, w zależności od parametrów. Uśrednione wartości z powtórzeń pomiarów.



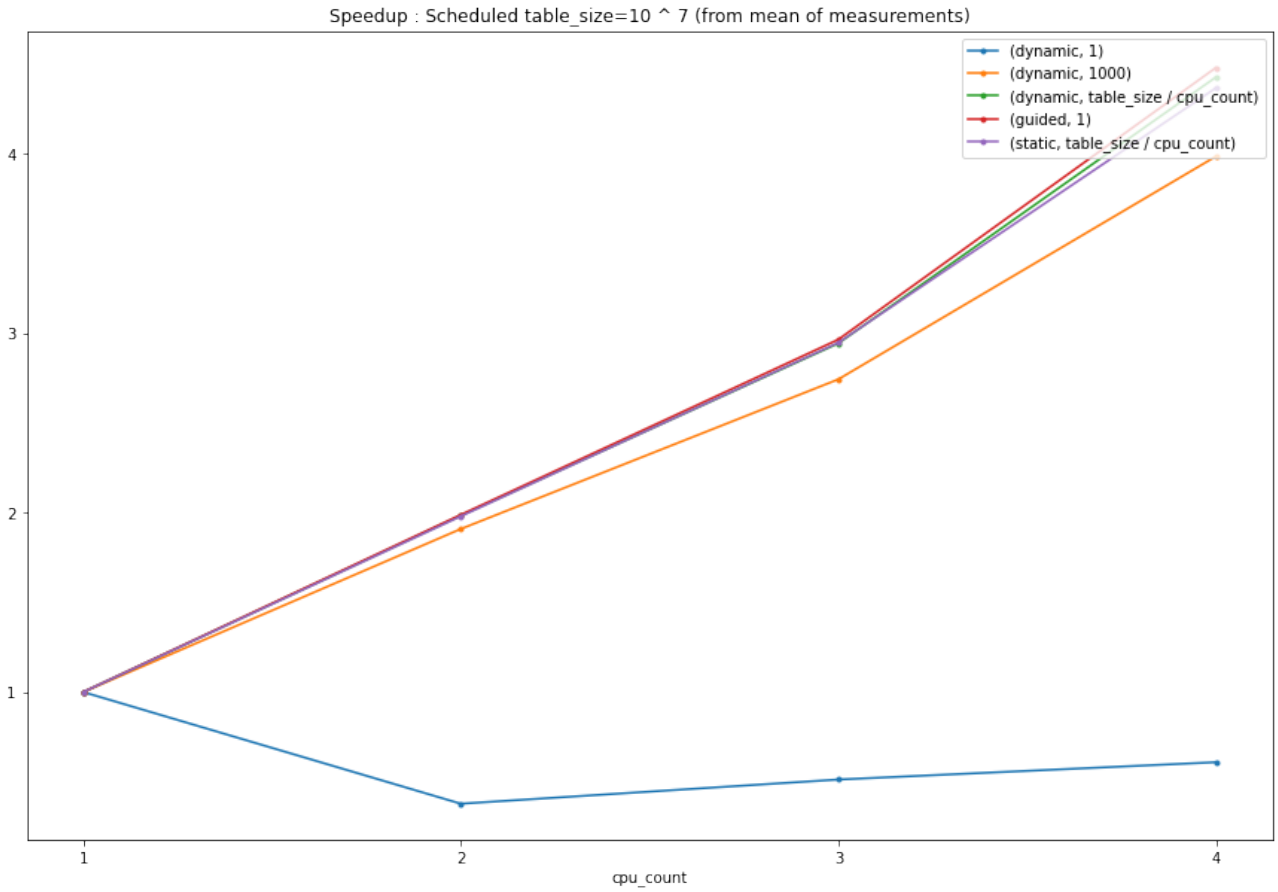


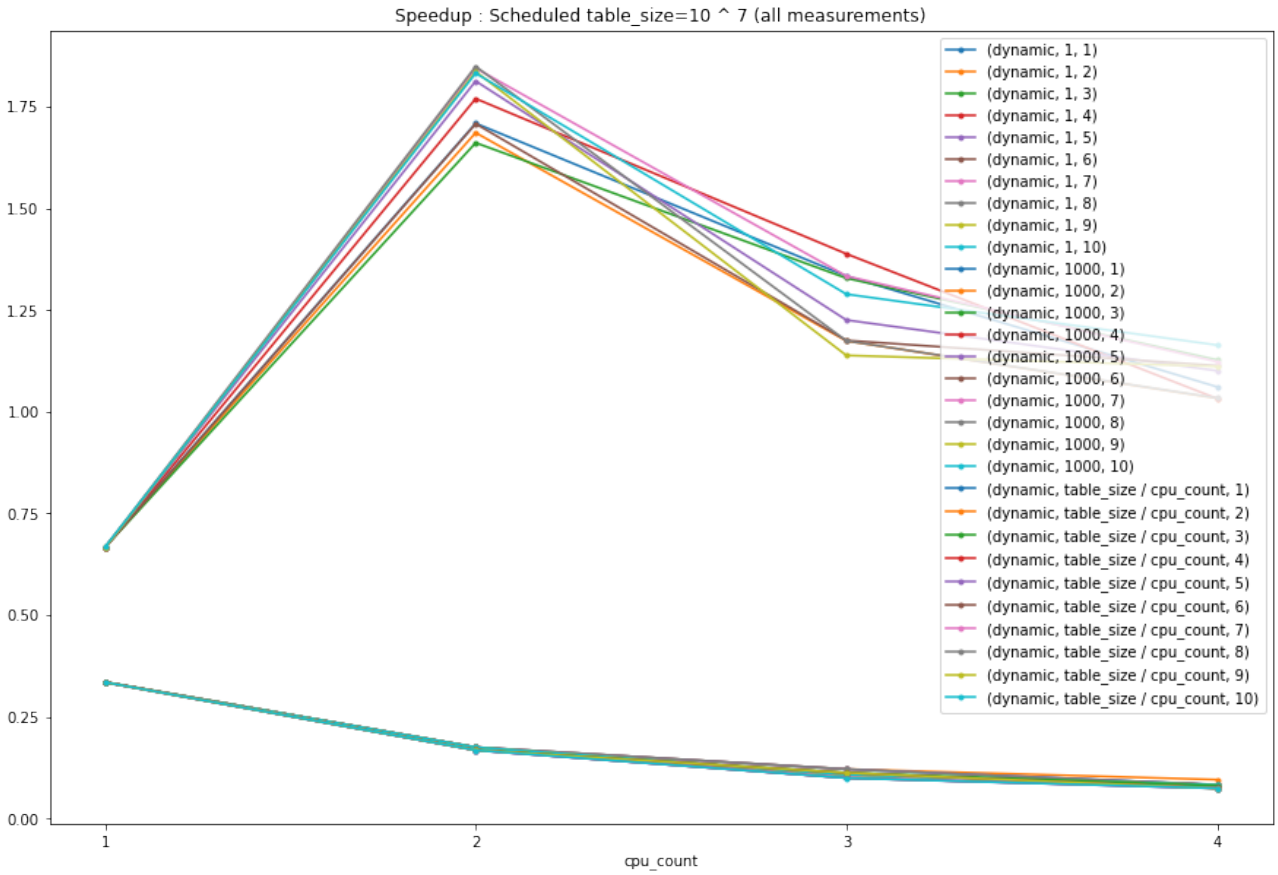
Rysunek 9: Pomiar czasu wykonania programu, dla problemu rozmiaru 10 ** 8, w zależności od parametrów. Uśrednione wartości z powtórzeń pomiarów.

Na podstawie zmierzonych czasów pracy przygotowaliśmy także wykresy przyśpieszenia programu.

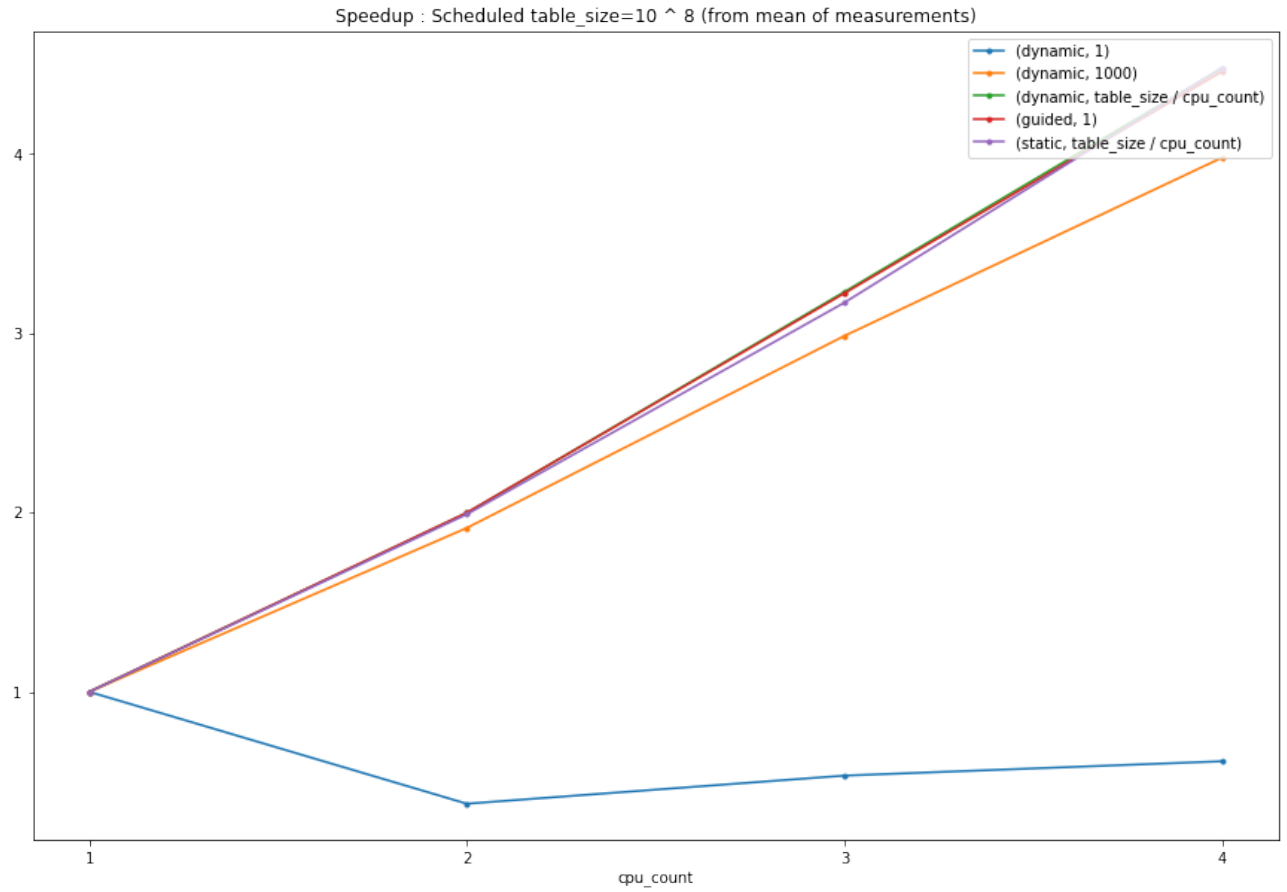


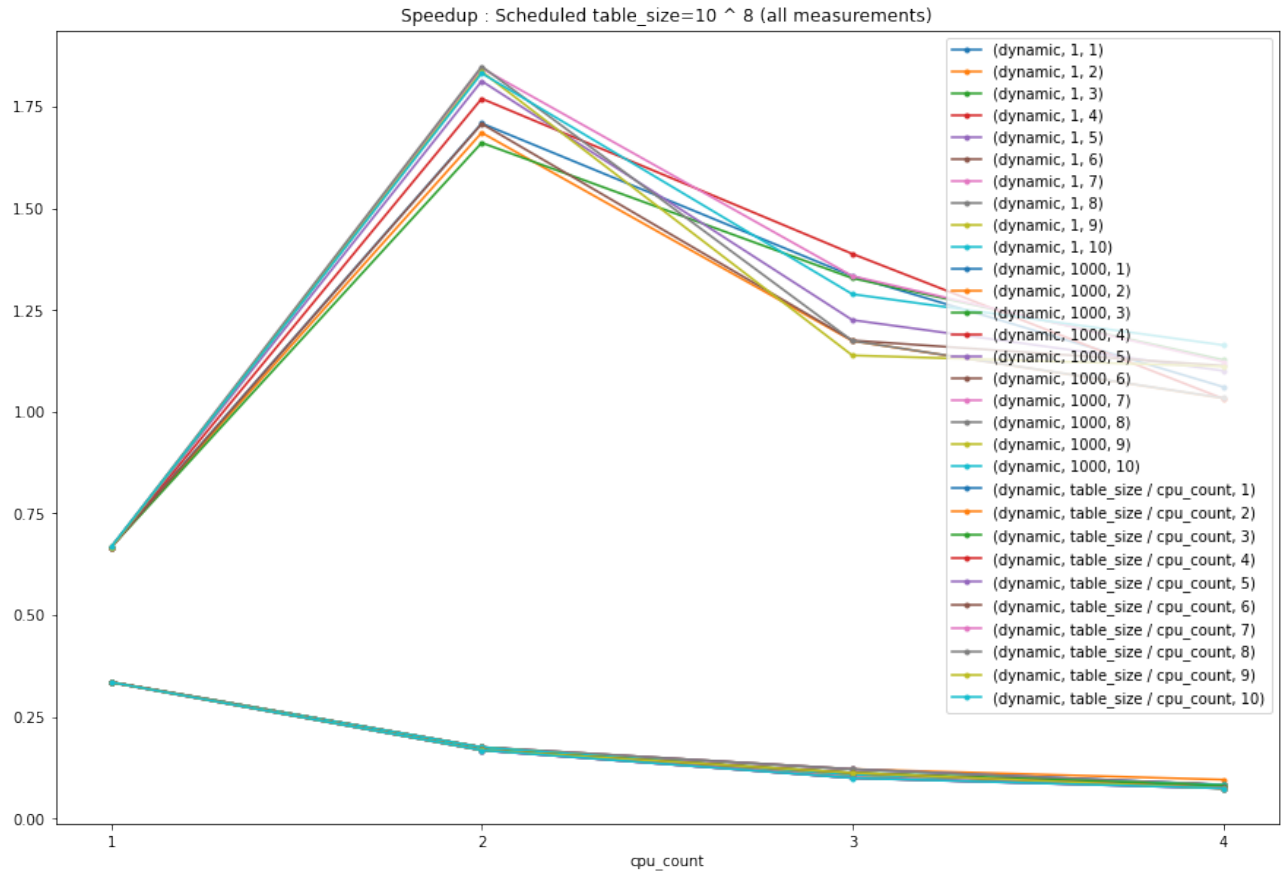
Rysunek 10: Pomiar przyśpieszenia czasu wykonania programu, dla problemu rozmiaru 10 ** 6, w zależności od parametrów. Uśrednione wartości z powtórzeń pomiarów.





Rysunek 11: Pomiar przyśpieszenia czasu wykonania programu, dla problemu rozmiaru 10 ** 7, w zależności od parametrów. Uśrednione wartości z powtórzeń pomiarów.

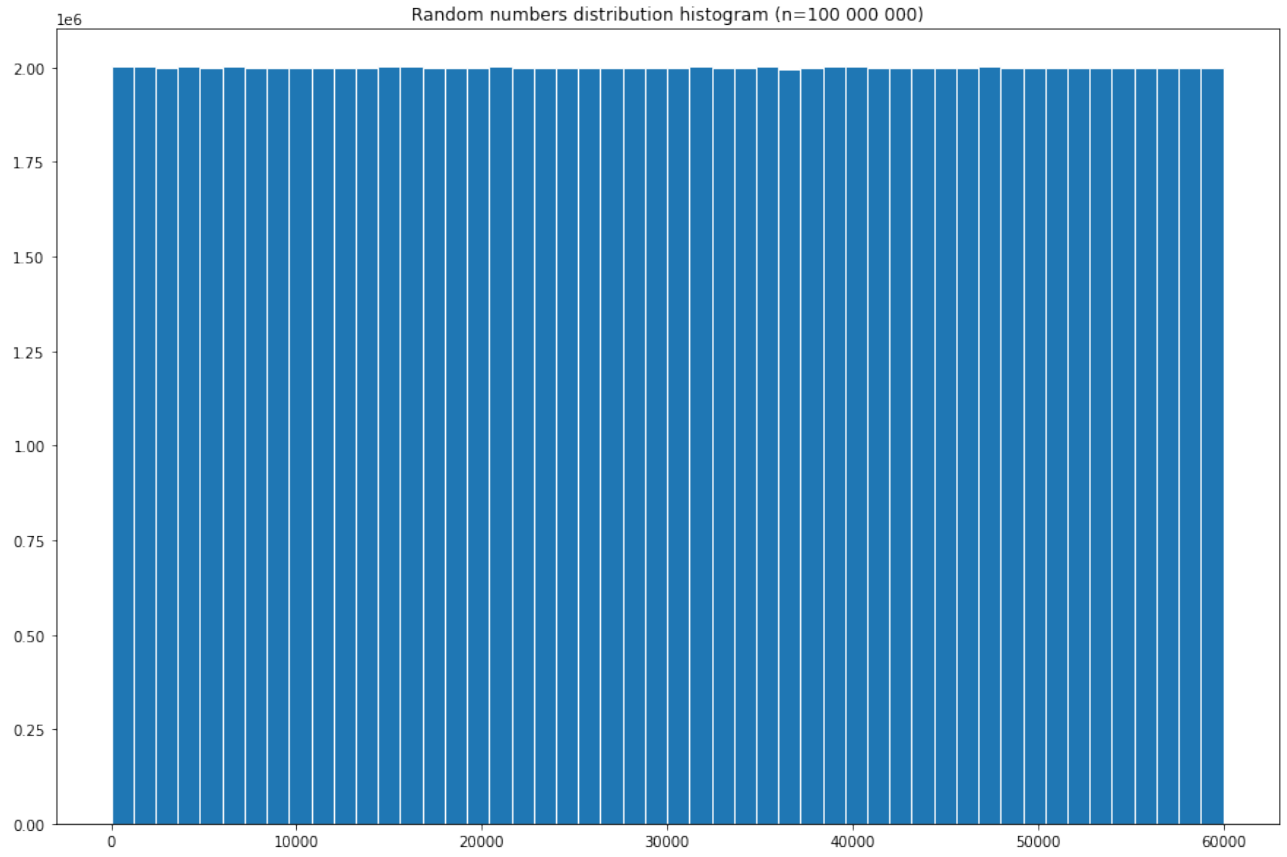




Rysunek 12: Pomiar przyśpieszenia czasu wykonania programu, dla problemu rozmiaru 10 ** 8, w zależności od parametrów. Uśrednione wartości z powtórzeń pomiarów.

1.3 Badanie rozkładu uzyskiwanych liczb pseudolosowych

Badanie poprawności danych wejściowych wobec założeń sortowania kubełkowego.
W celu zbadania rozkładu otrzymywanych z generatora liczb pseudolosowych przygotowany został duży zbiór liczb zwracanych przez generator, o rozmiarze 100000000. Uzyskane liczby zostały przedstawione na histogramie grupującym obserwacje w 100 kubełkach.



Rysunek 13: Histogram przedstawiający rozkład uzyskiwanych z generatora liczb pseudolosowych.

Na podstawie histogramu wnioskować możemy, że otrzymywany rozkład z generatora jest równomierny. Tak przygotowany generator dobrze sprawdzi się podczas testowania algorytmów sortujących.

1.4 Wnioski

Analizując otrzymane wyniki pomiarów, zauważyć możemy że każda z typów klauzuli *schedule* pozwoliła na uzyskanie zadowalających i porównywalnych przyśpieszeń, pod warunkiem wybrania odpowiedniej wartości parametru *chunk*. W szczególności, klauzula *guided* odpowiednio radzi sobie z jego dobraniem. Zauważyć możemy znaczny spadek wydajności, w przypadku wybrania zbyt małej wartości tego parametru dla wersji *dynamic*, skutkującego nawet obniżeniem wydajności programu pomimo większej liczby dostępnych procesorów.

1.5 Kod programu

```
1000 #include <omp.h>
     #include <stdio.h>
1002 #include <stdlib.h>

1004 #define MAX_RAND 1000;

1006 int main (int argc, char **argv) {
       int i, threads, array_size;
1008
       array_size = atoi(argv[1]);
1010   threads = atoi(argv[2]);

1012   omp_set_num_threads(threads);
```

```
  FILE *file;
  file = fopen("results.csv", "a");

  double time_start = omp_get_wtime();

  int *arr = (int*)calloc(array_size, sizeof(int));

  #pragma omp parallel shared(array_size) private(i)
  {
    unsigned int seed = omp_get_thread_num();

    #pragma omp for schedule(static)
    for(i=0; i<array_size; i++) {
      arr[i] = rand_r(&seed) % MAX_RAND;
    }
  }

  double time = omp_get_wtime() - time_start;

  fprintf(file, "static,%d,%d,%d,%f\n", array_size / threads, array_size, threads, time);
  fclose(file);

  return 0;
}
```

../ex3/schedule_static.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_RAND 1000;

int main (int argc, char **argv) {
  int i, threads, chunk_size, array_size;

  chunk_size = atoi(argv[1]);
  array_size = atoi(argv[2]);
  threads = atoi(argv[3]);

  omp_set_num_threads(threads);

  FILE *file;
  file = fopen("results.csv", "a");

  double time_start = omp_get_wtime();

  int *arr = (int*)calloc(array_size, sizeof(int));

  #pragma omp parallel shared(chunk_size, array_size) private(i)
  {
    unsigned int seed = omp_get_thread_num();

    #pragma omp for schedule(dynamic, chunk_size)
    for(i=0; i<array_size; i++) {
      arr[i] = rand_r(&seed) % MAX_RAND;
    }
  }

  double time = omp_get_wtime() - time_start;

  fprintf(file, "dynamic,%d,%d,%d,%f\n", chunk_size, array_size, threads, time);
  fclose(file);

  return 0;
}
```

../ex3/schedule_dynamic.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_RAND 1000;

int main (int argc, char **argv) {
```

```
    int i, threads, chunk_size, array_size;
1008
    chunk_size = atoi(argv[1]);
1010 array_size = atoi(argv[2]);
    threads = atoi(argv[3]);
1012
    omp_set_num_threads(threads);
1014
    FILE *file;
1016 file = fopen("results.csv", "a");

1018 double time_start = omp_get_wtime();

1020 int *arr = (int*)calloc(array_size, sizeof(int));

1022 #pragma omp parallel shared(chunk_size, array_size) private(i)
    {
1024   unsigned int seed = omp_get_thread_num();

1026   #pragma omp for schedule(guided, chunk_size)
      for(i=0; i<array_size; i++) {
1028     arr[i] = rand_r(&seed) % MAX_RAND;
      }
1030 }

1032 double time = omp_get_wtime() - time_start;

1034 fprintf(file, "guided,%d,%d,%d,%f\n", chunk_size, array_size, threads, time);
    fclose(file);
1036
    return 0;
1038 }
```

../ex3/schedule_guided.c

2 Implementacja równoległego algorytmu sortowania kubełkowego

2.1 Wstęp

W ramach ćwiczeń laboratoryjnych przygotowaliśmy wraz z kolegą z grupy implementację algorytmu sortowania kubełkowego w sposób równoległy dla tego samego sposobu generowania liczb pseudolosowych i struktury danych przechowujących kubełki na dwa różne sposoby. W implementacji kolegi każdy z wątków czytał całą tablicę wejściową i rozdzielał liczby do przypisanych do niego kubełków. W mojej implementacji, każdy z wątków czytał jedynie fragment tablicy i rozdzielał liczby do odpowiedniego kubełka (z synchronizacją na dostępie do kubełka, dostęp do tablicy nie musi być synchronizowany). Na potrzeby sprawozdania będę nazywał te implementacje odpowiednio wersją pierwszą i druga algorytmu.

Testy i pomiary czasowe wykonania algorytmu wykonane zostały na procesorze *Apple M1 Max* (chyba że wskazano inaczej) wykonanym w architekturze *ARM* o 10 rdzeniach roboczych, przy czym wskazać należy że procesor posiada 8 rdzeni wysokiej wydajności i 2 rdzenie energooszczędne co może zaburzać obserwowane wyniki przyśpieszenia.
Każdy z pomiarów wykonany został dziesięciokrotnie, a wyniki uśrednione.

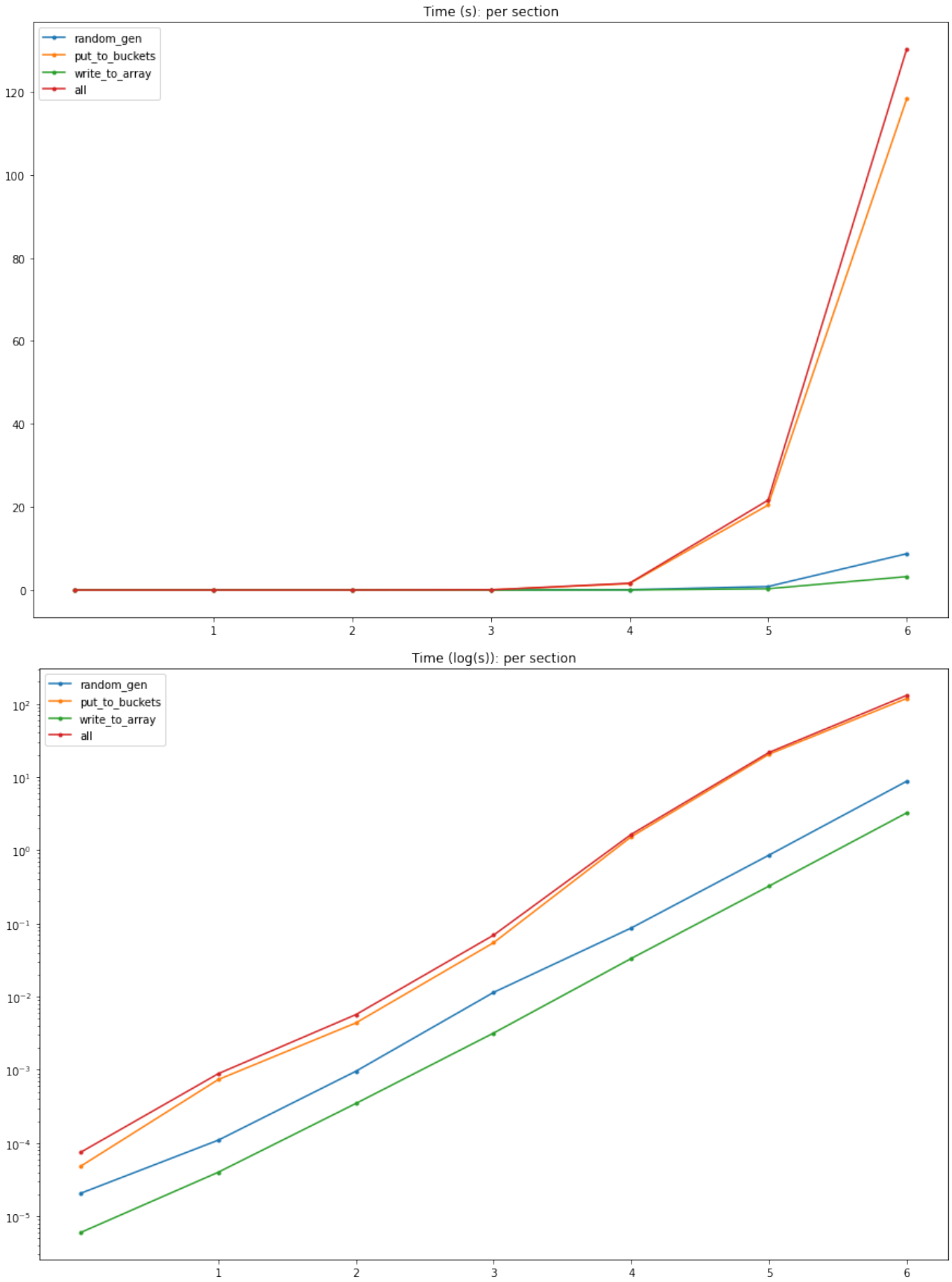
2.2 Struktura danych

W obu algorytmach do przechowywania kubełków wykorzystaliśmy dynamiczne tablice. Początkowo każda z tablic miała rozmiar równy podwojonej liczbie elementów tablicy podzielonej przez ilość kubełków. W przypadku gdy dodanie kolejnej wartości do tablicy spowodowałoby przekroczenie zaalokowanego rozmiaru tablicy, tworzony była tablica o dwukrotnie większym rozmiarze do której przepisywana była zawartość obecnego kubełka.

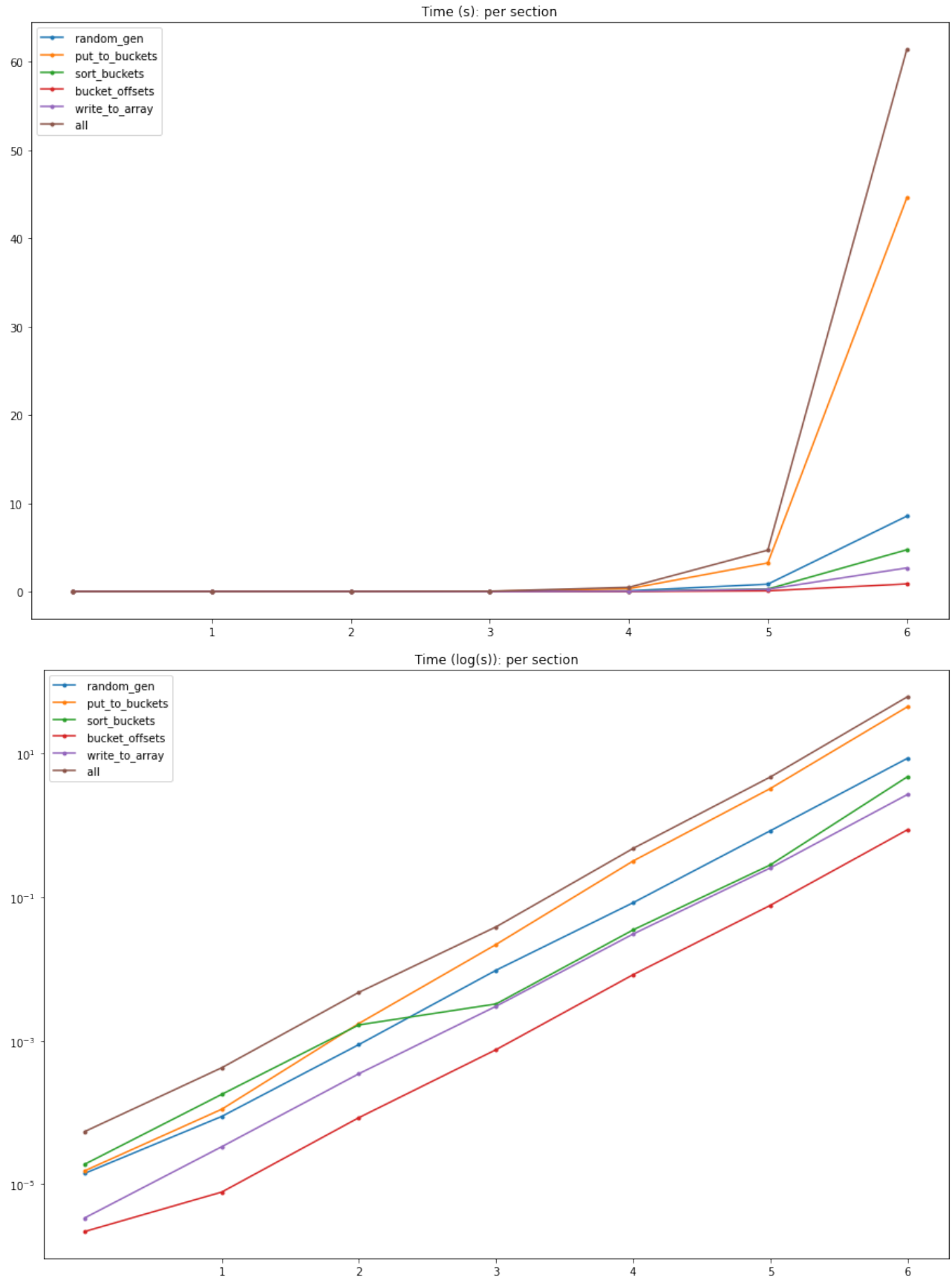
W celu analizy jak duży koszt obliczeniowy dla algorytmu stanowi przepisywanie kubełków (co jest kosztowne obliczeniowo zarówno przez konieczność przekopiowania starej tablicy do nowej, jak i alokacji pamięci), zliczyliśmy ilość przepisań. Dla tablicy o rozmiarze 1000000000, ilość przepisań podczas sortowania wyniosła 542997, czyli około $\frac{5}{10000}$ rozmiaru tablicy, co jest wartością pomijalną wobec złożoności obliczeniowej całego problemu.

2.3 Badanie zachowania algorytmu dla sortowania sekwencyjnego

Badając zależność czasu sekwencyjnego wykonania algorytmu w zależności od rozmiaru tablicy na wejściu zauważyć możemy, że w przypadku obu algorytmów najwięcej czasu zajmuje wstawianie elementów tablicy do odpowiednik kubełków. Jest to efekt oczekiwany, jako ze to właśnie na tym etapie odbywa się sortowanie tablicy. Wszystkie składowe algorytmu rosną tak samo, wraz ze wzrostem rozmiaru wejścia.



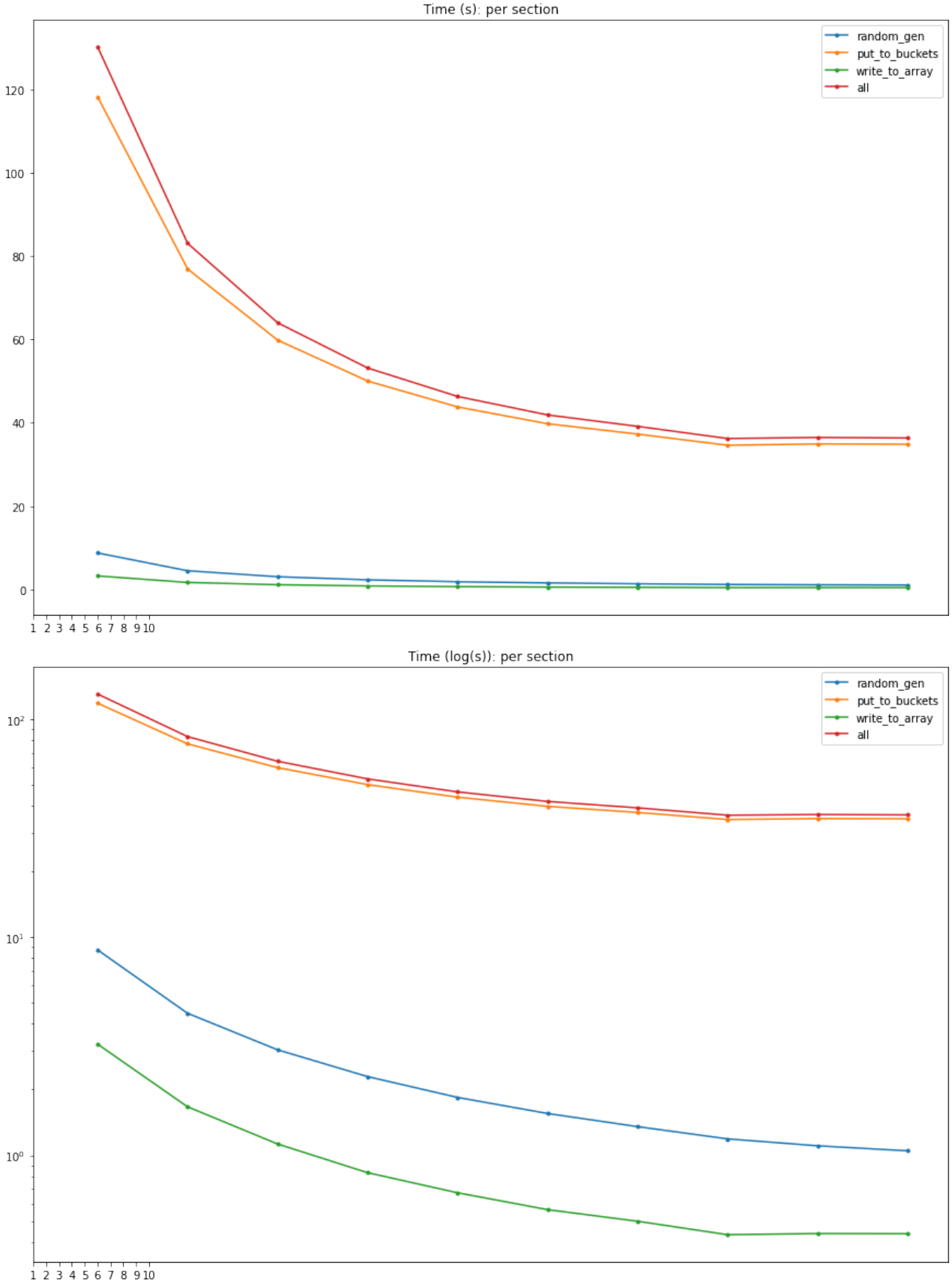
Rysunek 14: Pomiar czasu wykonania programu, w wersji pierwszej, wykonywanego sekwencyjnie, w zależności od rozmiaru tablicy (od 10^3 do 10^9).



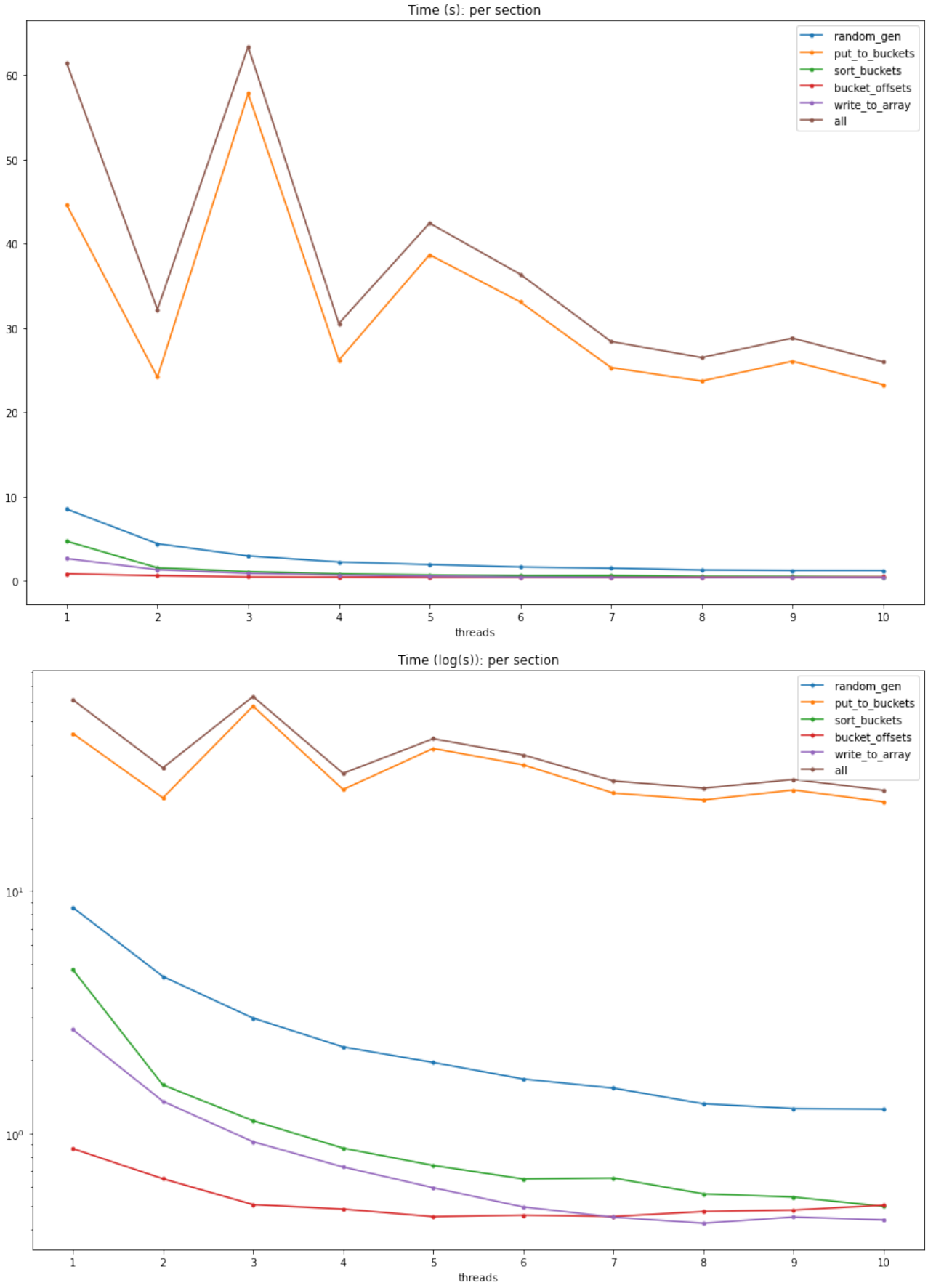
Rysunek 15: Pomiar czasu wykonania programu, w wersji drugiej, wykonywanego sekwencyjnie, w zależności od rozmiaru tablicy (od 10^3 do 10^9).

2.4 Badanie przyśpieszenia algorytmu

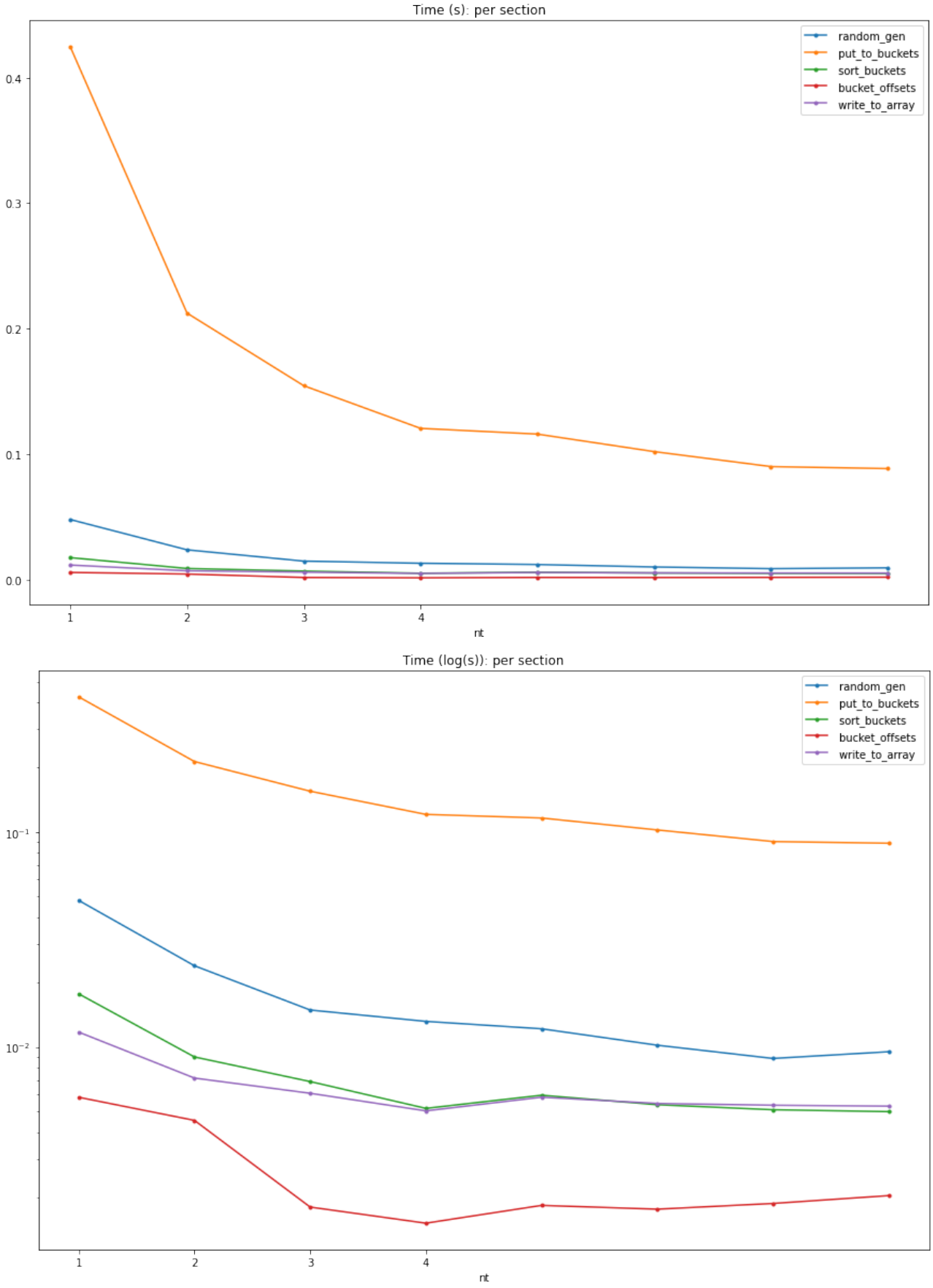
Analizując zmianę czasu wykonania programu w zależności od ilości rdzeni na których zrównoleglany był algorytm, zauważyć możemy że czas wykonania algorytmu w wersji pierwszej spada jednostajnie według oczekiwanej krzywej. W wersji drugiej, czas wykonania zmienia się skokowo w nieoczekiwany sposób. Jest to zaskakujący wynik, którego nie oczekiwałem, dlatego uruchomiłem program w tej wersji także na komputerze wyposażonym w procesor *Intel i7-8550u* z czterema rdzeniami roboczymi i ośmioma wątkami. W tym przypadku otrzymałem wyniki dużo bardziej zbliżone do oczekiwanych (z jedynym załamaniem wynikającym z hyperthreadingu), dlatego zaburzenia uznaję za spowodowane specyficzną architekturą procesora z którego początkowo korzystałem (architektura *ARM*, rdzenie o różnej wydajności).



Rysunek 16: Pomiar czasu wykonania programu, w wersji pierwszej, w zależności od ilości rdzeni, dla tablicy 10^9.

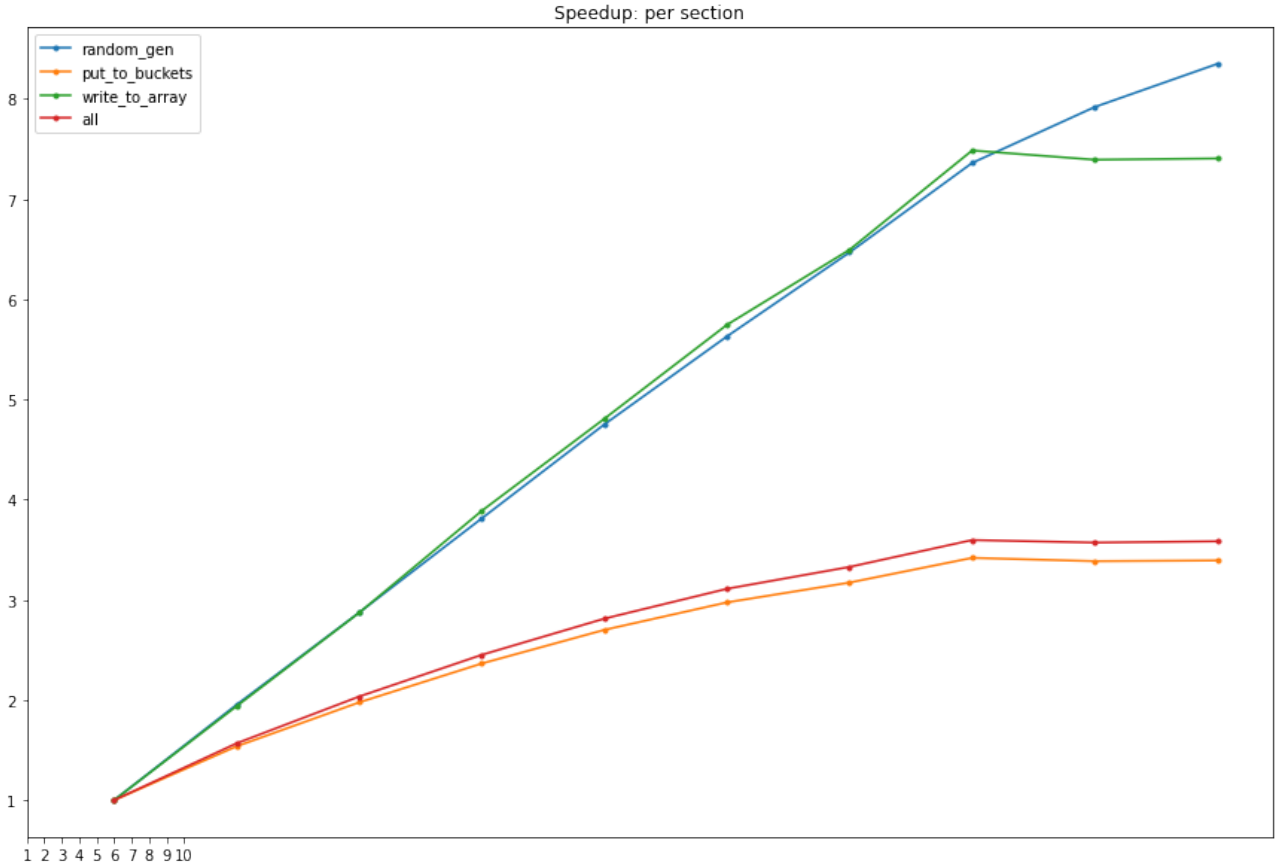


Rysunek 17: Pomiar czasu wykonania programu, w wersji drugiej, w zależności od ilości rdzeni, dla tablicy 10^9.

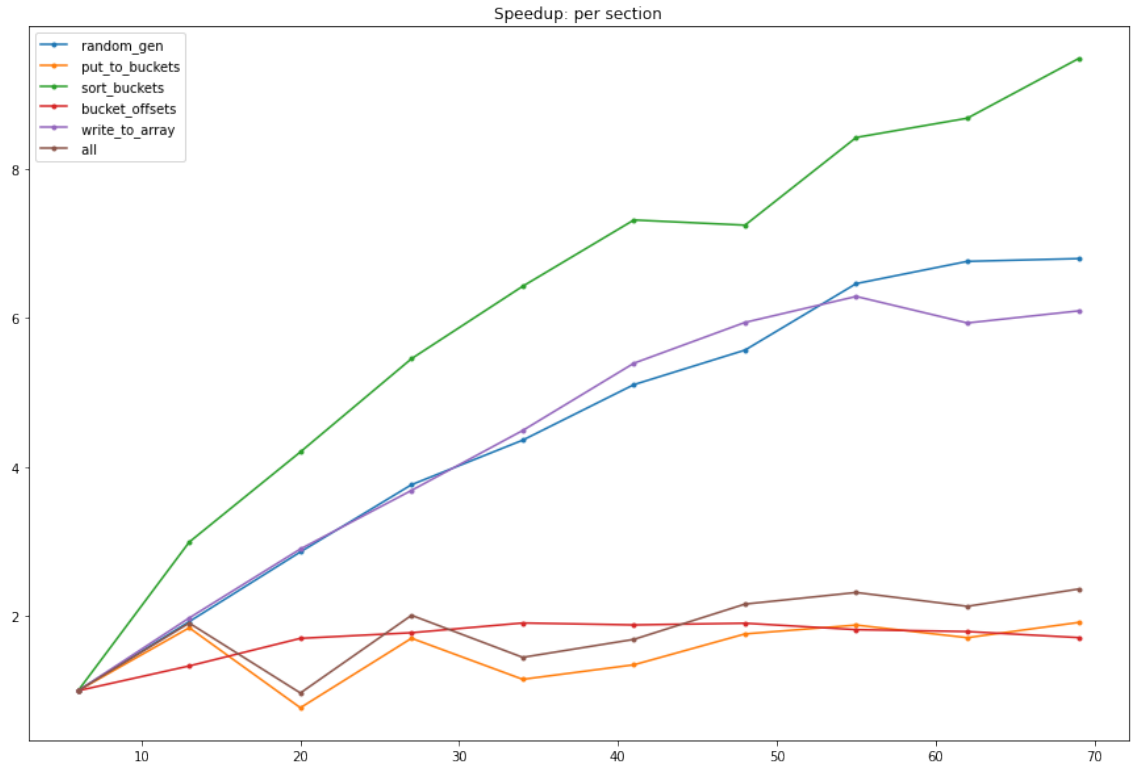


Rysunek 18: Pomiar czasu wykonania programu, w wersji drugiej, w zależności od ilości rdzeni, dla tablicy 10^9, uruchamianego na procesorze *Intel i7*.

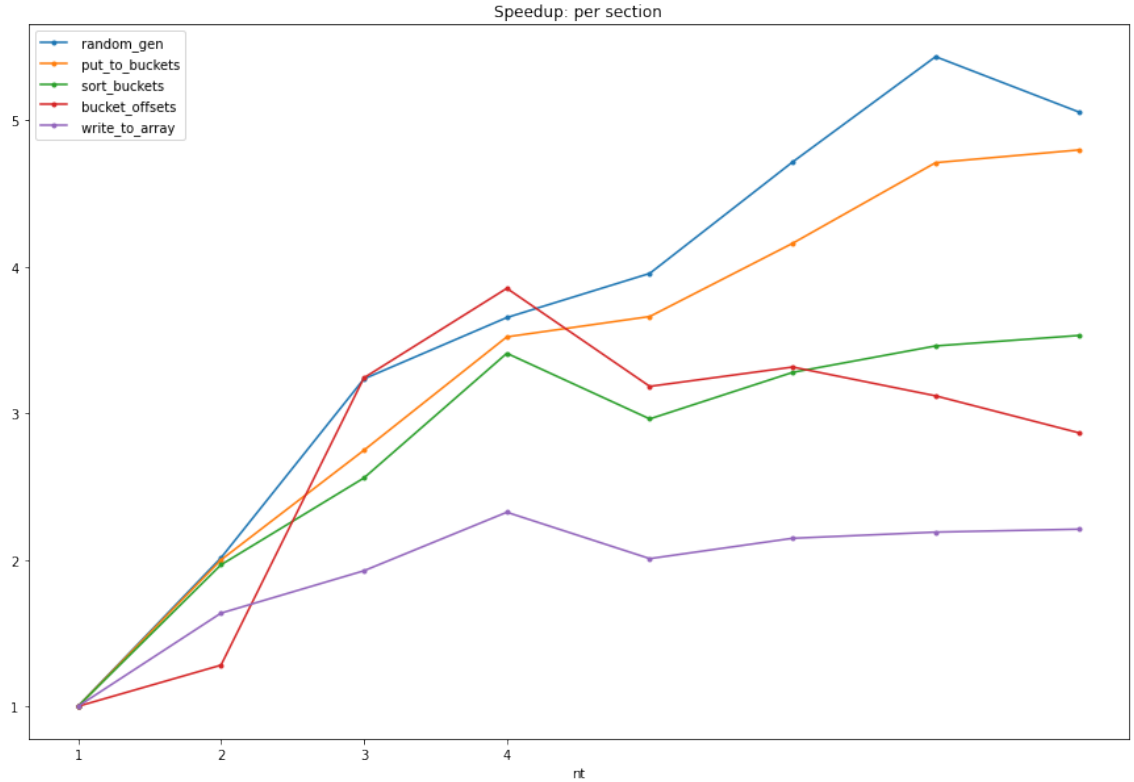
Następnie przygotowałem pomiary przyśpieszenia algorytmu. We wszystkich przypadkach program przyśpieszał znacząco wraz ze wzrostem ilości rdzeni. W przypadku wersji pierwszej, ponownie wykres najbardziej przypominał oczekiwany. W przypadku wersji drugiej uruchomionej na procesorze *i7* zauważyć możemy załamanie wynikające z *hyperthreadingu* po wzroście powyżej czterech wątków.



Rysunek 19: Pomiar przyśpieszenia programu, w wersji pierwszej, dla tablicy 10^9.



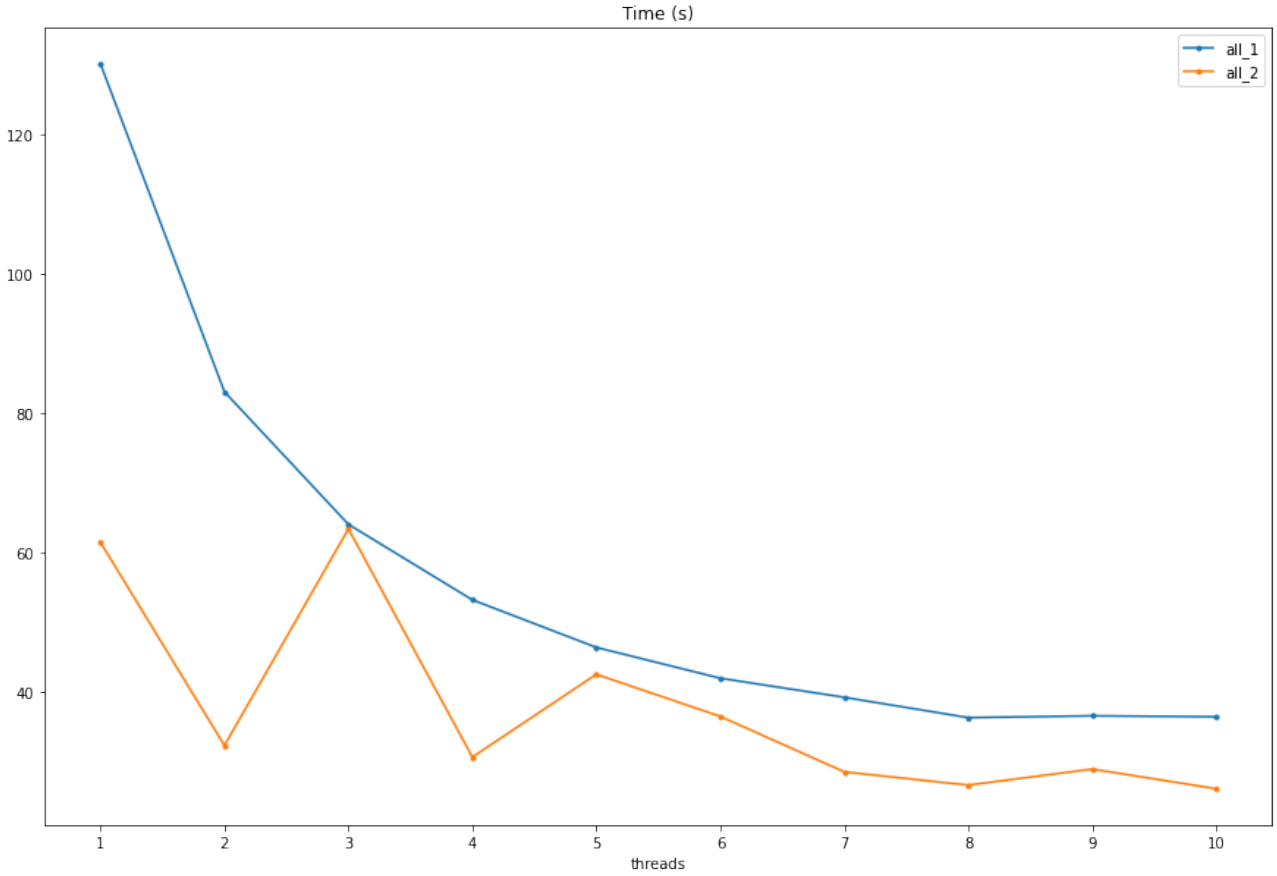
Rysunek 20: Pomiar przyśpieszenia programu, w wersji drugiej, dla tablicy 10^9.



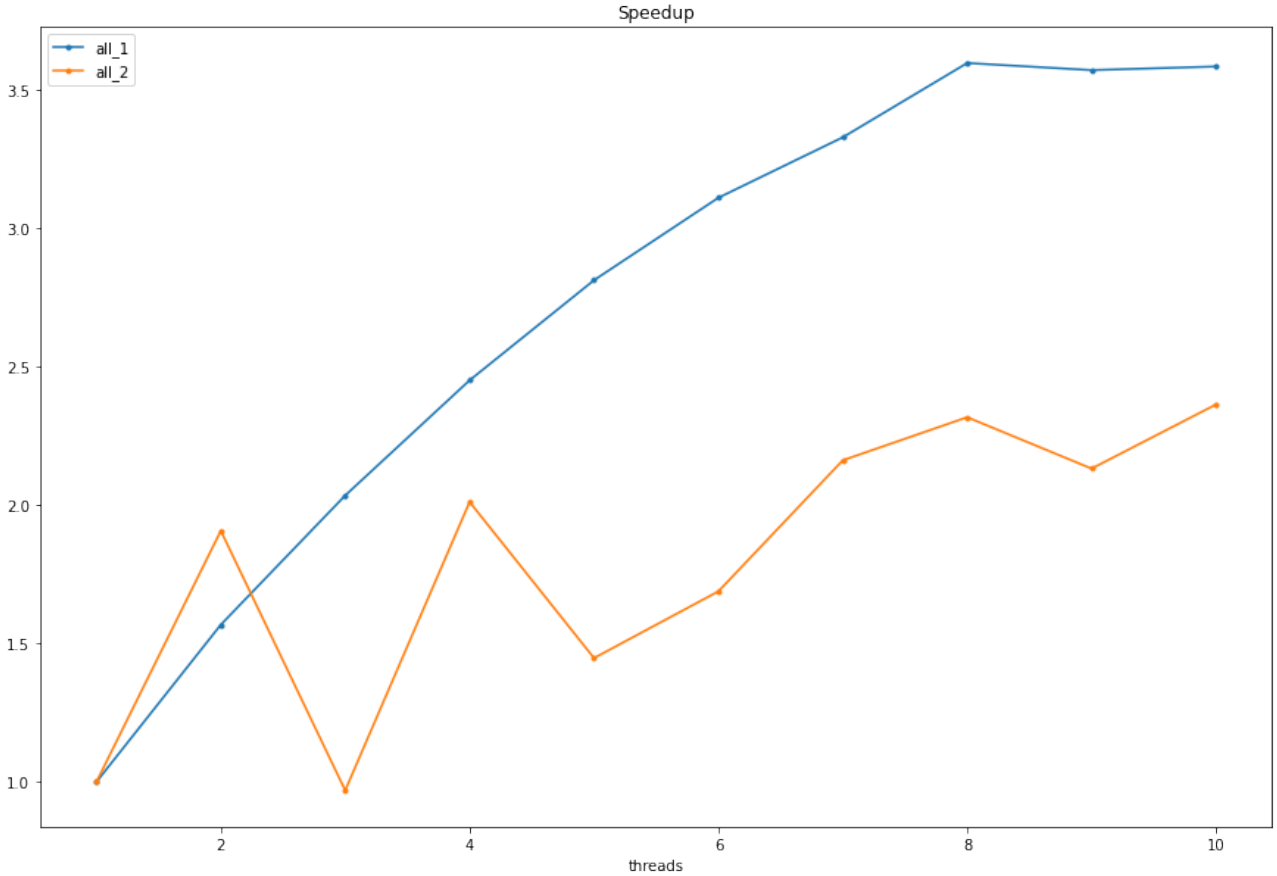
Rysunek 21: Pomiar przyśpieszenia programu, w wersji drugiej, dla tablicy 10^9, uruchamianego na procesorze *Intel i7*.

2.5 Porównanie implementacji

W celu porównania złożoności obliczeniowej i czasu wykonania obu wersji algorytmu równoległego przygotowałem wykres czasu wykonania i przyśpieszenia dla stałego rozmiaru tablicy wejściowej (wynoszącego 1000000000) w zależności od ilości rdzeni na których algorytm był zrównoleglany. Oba programy uruchamiane były na tym samym środowisku sprzętowym. Zaobserwować możemy dużo niższe czasy wykonania, niezależnie od ilości rdzeni na których program był uruchamiany, programu w wersji w której każdy z wątków czyta jedynie fragment tablicy wejściowej rozdzielając do kubełków. Przyśpieszenia procentowe są co do wartości mniejsze w wersji drugiej, co paradoksalnie może wynikać z mniejsze ilości obliczeń wykonywanych przez algorytm.



Rysunek 22: Pomiar czasu wykonania programu, w zależności od wersji i ilości rdzeni.



Rysunek 23: Pomiar przyśpieszenia, w zależności od wersji i ilości rdzeni.

2.6 Kod programu (wersja druga)

```
1000 #include <omp.h>
     #include <stdio.h>
1002 #include <stdlib.h>
     #include <string.h>
1004 #include <time.h>
     #include <limits.h>
1006
     // #define WRITE_FILE_IN true
1008 // #define WRITE_FILE_OUT true
     const int initial_bucket_size_mul = 2;
1010
     #define MAX_RAND 60000
1012 typedef unsigned long int COUNT;
     typedef unsigned short int VAL;
1014
     #define MAX(x, y) (((x) > (y)) ? (x) : (y))
1016 int bucket_rewrites_count = 0;

1018 VAL *rewrite_bucket(VAL *bucket, int current_size, int new_size) {
         VAL *new_bucket = calloc(new_size, sizeof(VAL));
1020     memcpy(new_bucket, bucket, current_size * sizeof(*bucket));
         free(bucket);
1022     // bucket_rewrites_count++;
         return new_bucket;
1024 }

1026 void append(VAL **bucket, int *bucket_size, int *items_count, VAL value) {
         if ((*items_count) + 5 >= (*bucket_size)){
1028         *bucket = rewrite_bucket(*bucket, *bucket_size, (*bucket_size) * 2);
             *bucket_size = (*bucket_size) * 2;
1030      }
         (*bucket)[*items_count] = value;
1032     (*items_count)++;
     }
1034
     void sort_bucket(VAL *bucket, int bucket_size) {
1036   int i = 0;
       for (i=1;i<bucket_size; i++) {
1038         VAL val = bucket[i];

1040         int j = i - 1;
             while(j >= 0 && bucket[j] > val){
1042             bucket[j + 1] = bucket[j];
                 j--;
1044         }
             bucket[j + 1] = val;
1046   }
     }
1048
     void write_bucket_to_array(VAL *bucket, int bucket_offset, int bucket_size, VAL *arr){
1050     int place_in_array = bucket_offset;
         for(int j=0; j<bucket_size; j++) {
1052       arr[place_in_array] = bucket[j];
           place_in_array++;
1054     }

1056 }

1058 void random_array(VAL *array, int array_size, int thread_count){
       int i;
1060   #pragma omp parallel num_threads(thread_count) private(i)
       {
1062     unsigned int seed = (omp_get_thread_num()+1) * omp_get_wtime();
         for(i=omp_get_thread_num();i<array_size;i+=thread_count)
1064       array[i] = rand_r(&seed) % MAX_RAND;
       }
1066 }

1068 void dump_array(VAL *array, int array_size, FILE *file){
       for(int i=0; i<array_size; i++)
1070     fprintf(file, "%d\n", array[i]);
     }
1072
```

```
1074 struct Buckets {
       int buckets_count;
1076   VAL **buckets;
       int *bucket_sizes;
1078   int *last_in_bucket;
       omp_lock_t *bucket_locks;
1080   int *bucket_offset;
     };
1082
     void free_buckets(struct Buckets buckets) {
1084   for (int i = 0; i< buckets.buckets_count; i++){
         free(buckets.buckets[i]);
1086     omp_destroy_lock(&(buckets.bucket_locks[i]));
       }
1088   free(buckets.buckets);
       free(buckets.bucket_sizes);
1090   free(buckets.last_in_bucket);
       free(buckets.bucket_locks);
1092   free(buckets.bucket_offset);
     }
1094
     void put_values_to_buckets(VAL *array, int array_size, int thread_count, struct Buckets
         buckets){
1096   int i;
       int buckets_count = buckets.buckets_count;
1098   int bucket_size = (MAX_RAND + buckets_count - 1) / buckets_count;
      #pragma omp parallel num_threads(thread_count) private(i) shared(bucket_rewrites_count)
1100  {
        for(i=omp_get_thread_num(); i<array_size; i+=thread_count) {
1102      int bucket = array[i] / bucket_size;
          omp_set_lock(&(buckets.bucket_locks[bucket]));
1104      append(&buckets.buckets[bucket], &(buckets.bucket_sizes[bucket]), &(buckets.
        last_in_bucket[bucket]), array[i]);
          omp_unset_lock(&(buckets.bucket_locks[bucket]));
1106    }
       }
1108
     }
1110
     void create_buckets(struct Buckets *buckets, int array_size, int buckets_count) {
1112   buckets->buckets_count = buckets_count;
       buckets->buckets = (VAL**)calloc(buckets_count, sizeof(VAL*));
1114   buckets->bucket_sizes = (int *)calloc(buckets_count, sizeof(int));
       buckets->bucket_locks = (omp_lock_t *)calloc(buckets_count, sizeof(omp_lock_t));
1116   buckets->last_in_bucket = (int*)calloc(buckets_count, sizeof(int));

1118   int initial_bucket_size = array_size * initial_bucket_size_mul / buckets_count;
       for(int i=0; i<buckets_count; i++) {
1120     buckets->bucket_sizes[i] = initial_bucket_size;
         buckets->buckets[i] = (VAL*)calloc(initial_bucket_size, sizeof(VAL));
1122     omp_init_lock(&(buckets->bucket_locks[i]));
       }
1124   buckets->bucket_offset = (int*)calloc(buckets_count, sizeof(int));
     }
1126
     void calculate_offsets(struct Buckets *buckets, int array_size, int thread_count) {
1128   int i;
       int *bucket_offset_part = (int*)calloc(buckets->buckets_count, sizeof(int));
1130  #pragma omp parallel num_threads(thread_count) private(i)
       {
1132     for (i=omp_get_thread_num(); i<buckets->buckets_count; i+=thread_count){
          bucket_offset_part[i] = buckets->last_in_bucket[i];
1134      if ((i - thread_count) >= 0) bucket_offset_part[i] += bucket_offset_part[i-thread_count
        ];
         }
1136   }
      #pragma omp parallel num_threads(thread_count) private(i)
1138  {
        for (i=omp_get_thread_num(); i<buckets->buckets_count; i+=thread_count){
1140      buckets->bucket_offset[i] = 0;
          for(int k = 0; k < thread_count ; k++) {
1142        if ( i - k >= 0) buckets->bucket_offset[i] += bucket_offset_part[i - k];
          }
1144      buckets->bucket_offset[i] -= buckets->last_in_bucket[i];
       }
1146  }
```

```
  free(bucket_offset_part);
}

void put_to_array(VAL* array, struct Buckets *buckets, int thread_count){
  int i;
  #pragma omp parallel num_threads(thread_count) private(i)
  {
    for(i=omp_get_thread_num(); i<buckets->buckets_count; i+=thread_count)
      write_bucket_to_array(buckets->buckets[i], buckets->bucket_offset[i], buckets->
    last_in_bucket[i], array);
  }

}

void check_order(VAL* array, int array_size){
    for (int i=1; i<array_size; i++){
    if(array[i - 1] > array[i]){
      printf("ERROR: The array is not sorted. %d preceeding %d\n", array[i - 1], array[i]);
      exit(EXIT_FAILURE);
    }
  }
}

void sort_buckets(struct Buckets *buckets, int thread_count){
  int i;
  #pragma omp parallel num_threads(thread_count) private(i)
  {
    for (i=omp_get_thread_num(); i<buckets->buckets_count; i+=thread_count)
      sort_bucket(buckets->buckets[i], buckets->last_in_bucket[i]);
  }
}

void measure(int array_size, int nt) {
  int buckets_num = MAX(array_size / 10, 1);

  // creating array to sort
  VAL *arr = (VAL*)calloc(array_size, sizeof(VAL));
  VAL *array = arr;

  // creating data structures
  struct Buckets buckets_store;
  create_buckets(&buckets_store, array_size, buckets_num);

  // Write random values to array
  double time_start = omp_get_wtime();
  double time_random_gen = omp_get_wtime();
  random_array(array, array_size, nt);
  time_random_gen = omp_get_wtime() - time_random_gen;

  // Log arr values to file
  #if (defined(WRITE_FILE_IN) && WRITE_FILE_IN == true) || (defined(WRITE_FILE_OUT) &&
    WRITE_FILE_OUT == true)
  char file_name[30];
  sprintf(file_name, "test_%d_%d.csv", array_size, nt);
  FILE *file = fopen(file_name, "w+");
  // FILE *file = fopen("test.csv", "w+");
  #endif

  #if (defined(WRITE_FILE_IN) && WRITE_FILE_IN == true)
  // fprintf(file, "before\n");
  dump_array(array, array_size, file);
  #endif

  // putting values to buckets
  double time_put_to_buckets = omp_get_wtime();
  put_values_to_buckets(array, array_size, nt, buckets_store);
  time_put_to_buckets = omp_get_wtime() - time_put_to_buckets;

  // Sort in buckets
  double time_sort_buckets = omp_get_wtime();
  sort_buckets(&buckets_store, nt);
  time_sort_buckets = omp_get_wtime() - time_sort_buckets;

  // Calculate offsets
  double time_bucket_offsets = omp_get_wtime();
  calculate_offsets(&buckets_store, array_size, nt);
```

```
    time_bucket_offsets = omp_get_wtime() - time_bucket_offsets;

    // putting values to original array
    double time_write_to_array = omp_get_wtime();
    put_to_array(array, &buckets_store, nt);
    time_write_to_array = omp_get_wtime()- time_write_to_array;
    double time_end = omp_get_wtime();
    // Write resulting array to file
    #if (defined(WRITE_FILE_OUT) && WRITE_FILE_OUT == true)
    fprintf(file, "\nafter\n");
    dump_array(arr, array_size, file);
    #endif

    #if (defined(WRITE_FILE_IN) && WRITE_FILE_IN == true) || (defined(WRITE_FILE_OUT) &&
      WRITE_FILE_OUT == true)
    fclose(file);
    #endif

    // Check if array is sorted
    check_order(array, array_size);

    // printf("%d, %f, %f, %f, %f, %f, %d, %f\n",array_size, time_random_gen,
      time_put_to_buckets, time_sort_buckets, time_bucket_offsets, time_write_to_array,
      bucket_rewrites_count, time_end - time_start  );
    printf("%d,%d,%f,%f,%f,%f,%f,%f\n",nt, array_size, time_random_gen, time_put_to_buckets,
      time_sort_buckets, time_bucket_offsets, time_write_to_array, time_end - time_start  );
    bucket_rewrites_count = 0;

    // Free memory
    free(arr);
    free_buckets(buckets_store);
}

int main(int argc, char **argv) {
    printf("threads,array_size,random_gen,put_to_buckets,sort_buckets,bucket_offsets,
      write_to_array,all\n");
    // printf("threads, random_gen, put_to_buckets, sort_buckets, bucket_offsets, write_to_array
      , rewrite_count, all\n");
    // for (int n = 0; n<10 ; n++)
    // for(int thread_count = 1; thread_count <= max_thread_count; thread_count++)
    //   for(int array_size = 100000000; array_size <= 1000000000; array_size*=10)
    //     measure(array_size, thread_count);


      const int repeat = 5;

      const int min_thread_count = 1;
      const int max_thread_count = 10;
      const int max_array_size = 1000000000;
      const int min_array_size = 1000;
      const int array_size_step = 10;

      // const int min_thread_count = 10;
      // const int max_thread_count = 10;
      // const int min_array_size = 1000000000;
      // const int max_array_size = 1000000000;
      // const int array_size_step = 10;


      for (int n = 0; n<repeat ; n++)
          for(int thread_count = min_thread_count; thread_count <= max_thread_count;
      thread_count++)
              for(int array_size = min_array_size; array_size <= max_array_size; array_size*=
      array_size_step)
                  measure(array_size, thread_count);

    return 0;
}
```

../lab3/main.c