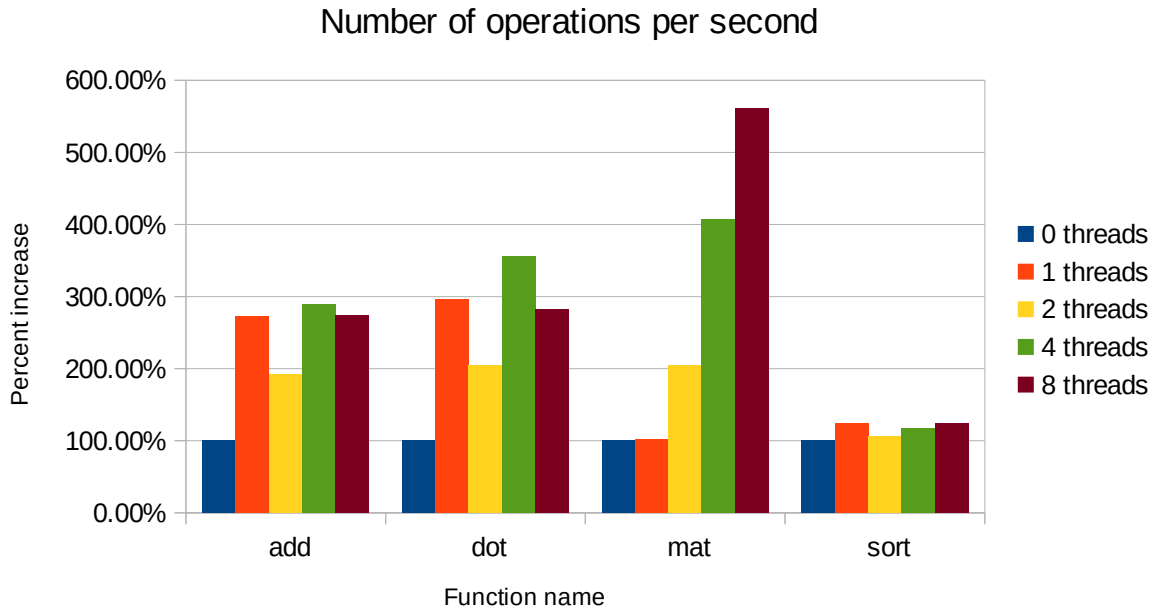
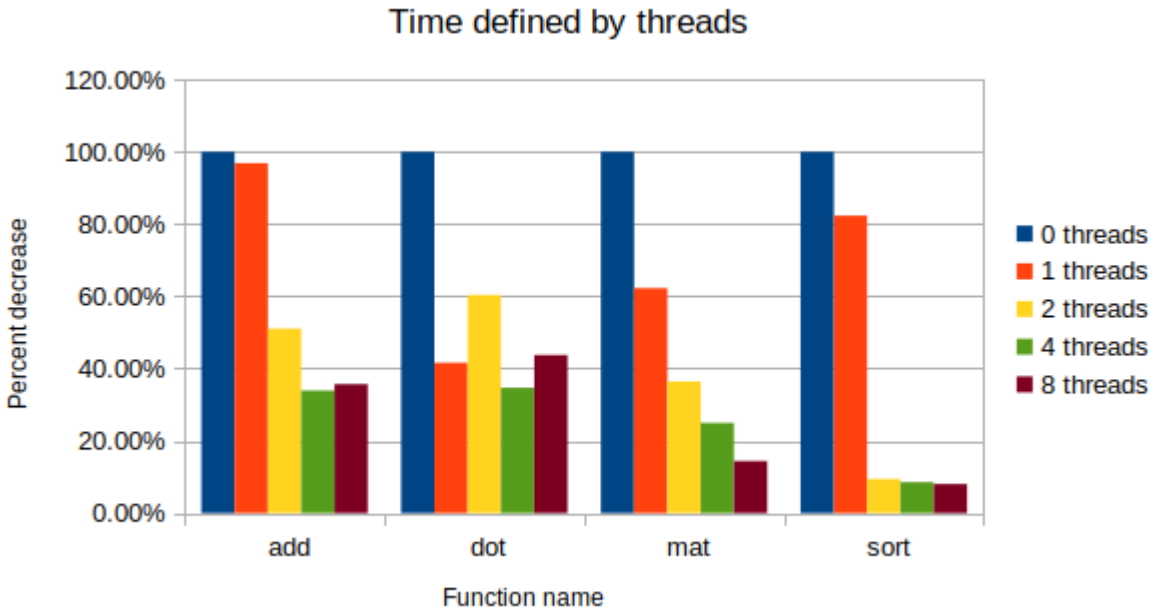
Poročilo N1 – Aleksander Grobelnik





V zgornji dveh razpredelnicah, razdeljeno na sredini, se lepo vidi časovne rezultate zbrane med izvajanjem našega programa.
Prva preglednica poimenovana “Number of operations per second” nam poda, kot nam že ime pove, izračunano število operacij, ki se izvedejo med našim izvajanjem. Navadna operacija, katero smo zgoraj zabeležili, je operacija, ki jo procesor izvede mnogokrat na sekunda in je sestavljenja iz množenja ali iz seštevanja oziroma odštevanja.
Klicali smo funckije s poimenovnjem “add” oziroma seštevanje dveh vektorjev po njunih medsebojnih elementih. Prav tako smo klicali funkcijo “dot” oziroma skalarno množenje dveh vektorjev, ki množi njuna istoležeča elementa. Sledi funkcija “mat”, ki je poimenovanje za matrični račun dveh matrik. Na koncu pa smo še obravnavali “sort” funckijo, ki sortira elemente v vektorju s

pomočjo metode deli in vladaj, ki razdeli vektor na pol, dokler lahko, vendar pri tem pazimo, da ne prekoračimo minimuma enega elementa.
Vse funkcije je bilo zanimivo paralelizirati, vendar nam zaradi omejitve števila niti, ne uspe najbolje optimizirati funkcije kot je sortiranje s pomočjo hitrega sortiranja. Če bi teoretično imeli neomejeno niti oziroma bi niti priklicali, v funkciji sami, bi lahko za vsak podproblem priklicali svojo nit, ki bi po potrebi poklicali tudi svoje niti. Tukaj moremo biti pazljivi, saj niti v običajnih sistemin ni neskončno mnogo in lahko pride do izgube podatkov, če smo neprevidni.
Na drugi preglednici poimenovana “Time defined by threads” lahko opazimo, da se nam časovna zahtevnost programa s povečanjem niti drastično upada, seveda dokler limita amdhalovega zakona, kjer pridemo do teoretičnega maksimuma in je nadaljevanje optimizacije s neskočno mnogo sredstvi enostavno nemogoča. Pridemo do tako imenovanega limita optimizacije, kar nam pove, da za morebitno pohitritev programa bomo morali pokledati kje drugje kot pa v predelu kode, ki je paralelizirana.
Če si pobližje pogledamo prva dva grafa vidimo nekaj izredno nenavadnega. Pri raličnih številih niti se nam program v povprečju bolje odreza kot bi pričakovali. Če bi vzeli samo te podatke bi lahko sklepali, da je program z 4 niti najhitrejši, vendar je v praksi drugače. Tukaj notri je slabo vkleščen variacije programa, ki vplivaja na rezultat za kar velik odstotek. Seveda pa to ni res pri “težji” programih/funkcijah za izračun matričnega produkta in pri sortirnem algoritmu “quicksort”. Tukaj lahko opazimo najprej drastični upad iz nič oziroma ene niti do dveh niti, kjer pa nato začne postopoma upadati. Tukaj se namreč začenjamo bližati teoretični možnosti optimizacije, in bi lahko opazili, da z bistvenim povečanjem niti ne bi niti dosti profitirali.

Koda programa:

```
#include <array>
#include <chrono>
#include <iostream>
#include <thread>
#include <vector>

using namespace std;
using namespace std::chrono;

typedef vector<double> dm1;
typedef vector<dm1> dm2;
// typedef vector<dm2> dm3;

const int count_threads = 8;

dm2 mat;
dm1 vec;
dm2 out_mat;
dm1 out_vec;

int rand_gen(int min, int max) {
  int range = max - min++;
  return rand() % range + min;
}

void fill_matrix(int size) {
  for (int j = 0; j < size; j++) {
    dm1 vec1;
    for (int k = 0; k < size; k++) {
```

```
      vec1.push_back(rand_gen(0, 10));
    }
    mat.push_back(vec1);
  }
}

void fill_vector(int size) {
  for (int i = 0; i < size; i++) {
    vec.push_back(rand_gen(0, 10));
  }
}

void fill_vector_zeros(int size) {
  for (int i = 0; i < size; i++) {
    out_vec.push_back(0);
  }
}

void fill_matrix_zeros(int size) {
  for (int i = 0; i < size; i++) {
    dm1 vec1;
    for (int j = 0; j < size; j++) {
      vec1.push_back(0);
    }
    out_mat.push_back(vec1);
  }
}

void print_matrix() {
  for (int i = 0; i < mat.size(); i++) {
    for (int j = 0; j < mat[i].size(); j++) {
      cout << mat[i][j] << " ";
    }
    cout << endl;
  }
  cout << endl;
}

void print_matrix_out() {
  for (int i = 0; i < out_mat.size(); i++) {
    for (int j = 0; j < out_mat[i].size(); j++) {
      cout << out_mat[i][j] << " ";
    }
    cout << endl;
  }
  cout << endl;
}

void print_vector() {
  for (int i = 0; i < vec.size(); i++) {
    cout << vec[i] << " ";
  }
```

```
  cout << endl;
}

void print_vector_out() {
  for (int i = 0; i < out_vec.size(); i++) {
    cout << out_vec[i] << " ";
  }
  cout << endl;
}

void add_vector(int s, int e) {
  for (int i = s; i < e; i++) {
    out_vec[i] = vec[i] + vec[i];
  }
}

void dot_product_vector(int s, int e) {
  for (int i = s; i < e; i++) {
    out_vec[i] = vec[i] * vec[i];
  }
}

void product_matrix(int s, int e) {
  for (int i = s; i < e; i++) {
    for (int j = 0; j < mat[i].size(); j++) {
      for (int k = 0; k < mat.size(); k++) {
        out_mat[i][j] += mat[i][k] * mat[k][j];
      }
    }
  }
}

int split(int a, int b) {
  int pivot = vec[a];
  int li = a;
  int ri = b;
  while (li < ri) {
    while (vec[li] <= pivot && li < b) {
      li++;
    }
    while (vec[ri] >= pivot && ri > a) {
      ri--;
    }
    if (li < ri) {
      swap(vec[li], vec[ri]);
    }
  }
  swap(vec[a], vec[ri]);
  return ri;
}

void quicksort(int a, int b) {
```

```
  if (a < b) {
    int splitter = split(a, b);
    quicksort(a, splitter - 1);
    quicksort(splitter + 1, b);
  }
}

int main(int argc, char **argv) {
  if (argc != 4) {
    cout << "Wrong number of given arguments" << endl;
    return 1;
  }
  auto size = stoi(argv[1]);
  auto seed = stoi(argv[2]);
  auto method = stoi(argv[3]);
  srand(seed);
  time_point<system_clock> start, end;

  cout << "computing program.." << endl;
  switch (method) {
  case 0: {
    fill_vector(size);
    fill_vector_zeros(size);
    array<thread, count_threads> threads;
    int offset = size / count_threads;
    int thread_size = sizeof(threads) / sizeof(thread);
    start = system_clock::now();
    for (int i = 0; i < thread_size; i++) {
      if (i == 0) {
        threads[i] = thread(add_vector, 0, offset);
      } else if (i == thread_size - 1) {
        threads[i] = thread(add_vector, i * offset + 1, size);
      } else {
        threads[i] = thread(add_vector, i * offset + 1, (i + 1) * offset);
      }
      // threads[i].join();
    }
    for (auto &t : threads) {
      t.join();
    }
    break;
  }
  case 1: {
    fill_vector(size);
    fill_vector_zeros(size);
    array<thread, count_threads> threads;
    int offset = size / count_threads;
    int thread_size = sizeof(threads) / sizeof(thread);
    start = system_clock::now();
    for (int i = 0; i < thread_size; i++) {
      if (i == 0) {
        threads[i] = thread(dot_product_vector, 0, offset);
```

```
    } else if (i == thread_size - 1) {
      threads[i] = thread(dot_product_vector, i * offset + 1, size);
    } else {
      threads[i] =
          thread(dot_product_vector, i * offset + 1, (i + 1) * offset);
    }
    //threads[i].join();
  }
  for (auto &t : threads) {
    t.join();
  }
  break;
}
case 2: {
  fill_matrix(size);
  fill_matrix_zeros(size);
  array<thread, count_threads> threads;
  int offset = size / count_threads;
  int thread_size = sizeof(threads) / sizeof(thread);
  start = system_clock::now();
  for (int i = 0; i < thread_size; i++) {
    if (i == 0) {
      threads[i] = thread(product_matrix, 0, offset);
    } else if (i == thread_size - 1) {
      threads[i] = thread(product_matrix, i * offset + 1, size);
    } else {
      threads[i] = thread(product_matrix, i * offset + 1, (i + 1) * offset);
    }
    //threads[i].join();
  }
  for (auto &t : threads) {
    t.join();
  }
  break;
}
case 3: {
  fill_vector(size);
  array<thread, count_threads> threads;
  int offset = size / count_threads;
  int thread_size = sizeof(threads) / sizeof(thread);
  start = system_clock::now();
  for (int i = 0; i < thread_size; i++) {
    if (i == 0) {
      threads[i] = thread(quicksort, 0, offset);
    } else if (i == thread_size - 1) {
      threads[i] = thread(quicksort, i * offset + 1, size);
    } else {
      threads[i] = thread(quicksort, i * offset + 1, (i + 1) * offset);
    }
    //threads[i].join();
  }
  for (auto &t : threads) {
```

```
      t.join();
    }
    quicksort(0, vec.size() - 1);
    break;
  }
  }
  end = system_clock::now();
  //print_vector();
  //print_vector_out();
  // print_vector(ret_dm2);
  //print_matrix();
  //print_matrix_out();
  cout << "method used: "
       << (method == 0   ? "add"
           : method == 1 ? "dot"
           : method == 2 ? "mat"
                         : "sort")
       << endl;

  duration<double> elapsed_seconds;
  elapsed_seconds = end - start;
  cout << "Elapsed time: " << elapsed_seconds.count() * 1000 * 1000 << "µs"
       << endl;
  cout << "Operations per second: "
       << size * (method != 2 ? 1 : size) / elapsed_seconds.count() << endl;
  cout << "Number of operations: " << size * (method != 2 ? 1 : size) << endl;
  return 0;
}
```

V spodni preglednici tudi dodajam neobdelane podatke iz katerih sem naredil zgornja dva grafa.

in IPS	add	dot	mat	sort	in µs
	204571000	188827000	56048.1	70270.2	
0 threads	100.00%	100.00%	100.00%	100.00%	
1 threads	272.54%	296.21%	102.32%	124.39%	
2 threads	191.91%	204.14%	204.52%	105.91%	
4 threads	289.02%	355.41%	407.64%	117.03%	
8 threads	274.57%	281.97%	560.96%	124.55%	

in µs	add	dot	mat	sort
0 threads	4988.28	4295.85	17529100	14230800
1 threads	100.00%	100.00%	100.00%	100.00%
2 threads	96.89%	41.62%	62.29%	82.36%
4 threads	51.06%	60.39%	36.42%	9.44%
8 threads	33.91%	34.69%	24.97%	8.54%
	35.69%	43.72%	14.40%	8.03%

0th

```
method used: add
Elapsed time: 4888.28 µs
```

Operations per second: 2.04571e+08
Number of operations: 1000000

method used: dot
Elapsed time: 4295.85 µs
Operations per second: 1.88827e+08
Number of operations: 1000000

method used: mat
Elapsed time: 17529100 µs
Operations per second: 57048.1
Number of operations: 1000000

method used: sort
Elapsed time: 1423080 µs
Operations per second: 70270.2
Number of operations: 1000000

1th

method used: add
Elapsed time: 1793.61µs
Operations per second: 5.57534e+08
Number of operations: 1000000

method used: dot
Elapsed time: 1787.88µs
Operations per second: 5.59323e+08
Number of operations: 1000000

computing program..
method used: mat
Elapsed time: 2.49195e+06µs
Operations per second: 401293
Number of operations: 1000000

method used: sort
Elapsed time: 1.14401e+06µs
Operations per second: 87411.7
Number of operations: 1000000

2th

method used: add
Elapsed time: 2547.13µs
Operations per second: 3.92599e+08
Number of operations: 1000000

method used: dot
Elapsed time: 2594.23µs
Operations per second: 3.85471e+08
Number of operations: 1000000

method used: mat
Elapsed time: 6.38443e+06µs
Operations per second: 156631
Number of operations: 1000000

method used: sort
Elapsed time: 1.34367e+06µs
Operations per second: 74422.9
Number of operations: 1000000

4th

method used: add
Elapsed time: 1691.33 µs
Operations per second: 5.91252e+08
Number of operations: 1000000

method used: dot
Elapsed time: 1490.08 µs
Operations per second: 6.71103e+08
Number of operations: 1000000

method used: mat
Elapsed time: 4376870 µs
Operations per second: 228474
Number of operations: 1000000

method used: sort
Elapsed time: 1215950 µs
Operations per second: 82240.2
Number of operations: 1000000

8th

method used: add
Elapsed time: 1780.33µs
Operations per second: 5.61695e+08
Number of operations: 1000000

method used: dot
Elapsed time: 1878.13 µs
Operations per second: 5.32444e+08
Number of operations: 1000000

method used: mat
Elapsed time: 2524180 µs
Operations per second: 396169
Number of operations: 1000000

method used: sort
Elapsed time: 1142550 µs

Operations per second: 87523.5
Number of operations: 1000000