Факултет техничких наука

Универзитет у Новом Саду

Рачунарски системи високих перформанси

Секвенцијална и паралелна имплементација мрављег алгоритма за решавање проблема трговачког путника

Аутор:
Алекса Ђукић

Индекс:
E2 84/2024

15. август 2025.

Сажетак

У овом раду разматра се проблем трговачког путника и његово решавање применом мрављег алгоритма. Алгоритам је имплементиран у програмском језику Ц, прво секвенцијално, затим помоћу *OpenMP* директива за паралелизацију, и на крају употребом *CUDA* интерфејса за извршавање на графичким процесорима. Посебна пажња посвећена је изазовима паралелизације, синхронизације и оптимизације приступа меморији. Експерименти су спроведени на скуповима различитих величина, а резултати показују да *OpenMP* имплементација постиже приметна убрзања на вишејезгарним системима, док *CUDA* верзија обезбеђује значајне добитке у перформансама код великих инстанци проблема, захваљујући масовној паралелизацији и ефикасном коришћењу ресурса графичког процесора.

Садржај

Списак изворних кодова

Списак слика

Списак табела

1 Увод

Проблем трговачког путника је класичан проблем у комбинаторној оптимизацији, који поставља питање: "Ако имамо листу градова и удаљеност између сваког пара градова, који ја најкраћи пут којим можемо обићи све градове и вратити се у град из којег смо кренули?". Овај, проблем је добро познат у теорији рачунарских наука и спада у класу *NP-hard* проблема [4]. Постоје бројна алгоритамска решења овог проблема, од којих већина нуди апроксимативна решења са разумним временима извршавања, док су егзактна решења погодна за коришћење само за ограничен број градова.

У овом раду као метод решавања биће примењен мрављи алгоритам (engl. *Ant Colony Optimization*, ACO) [1], који постиже апроксимативно решење уз бољу временску сложеност у односу на наивни приступ, а такође оставља простор за скраћивање времена извршавања техникама паралелизације. Имплементација алгоритма изведена је у Ц програмском језику, а за побољшање перформанси коришћени су *OpenMP* [3] и *CUDA* [2] који представљју изузетно ефикасне алате за развијање конкурентних програма.

2 Мрављи алгоритам

Мрављи алгоритам је пробабилистичка техника решавања проблема који се могу свести на проналажење добрих путања за обилазак графа. Алгоритам је инспирисан понашањем правих мрава, тј. њиховим начином комуникације преко феромона.

2.1 Опис и мотивација алгоритма

Алгоритам започиње тиме што се одређени број мрава поставља у различите градове (чворове на графу), и сваки од њих бира своју путању обиласка свих осталих градова. Ова путања заснована је на случајном одабиру, где се шансе да мрав одабере неки град као следећи у својој путањи рачуна по формули:

$$p_{xy}^k = \frac{(\tau_{xy}^\alpha)(\eta_{xy}^\beta)}{\sum_{z \in allowed_y}(\tau_{xy}^\alpha)(\eta_{xy}^\beta)} \tag{1}$$

Где p_{xy}^k означава шансе да мрав k пређе из града x у град y. Симбол τ_{xy}^α представља количину феромона депонованог на путањи између града x и y, а η_{xy}^β пожељност путање између ових градова, која је обично једнака реципрочној вредности њихове удаљености. Параметри $\alpha \geq 0$ и $\beta \geq 1$ одлучују о количини утицаја који феромони и пожељност имају на одабир следеће дестинације.

Након што једна генерација мрава заврши своје путање, ажурирају се вредности феромона на путањама (ивицама графа) којима су пролазили по следећој формули:

$$\tau_{xy} \leftarrow (1-\rho)\tau_{xy} + \sum_k^m \Delta\tau_{xy}^k \tag{2}$$

Где је τ_{xy} количина феромона депонованог на путању између градова x и y, ρ је коефицијент испаравања феромона, а $\Delta\tau_{xy}^k$ је количина феромона коју је депоновао мрав k. У случају проблема трговачког путника ова количина рачуна се на следећи начин:

$$\Delta\tau_{xy}^k = \begin{cases} Q/L_k & \text{ако мрав } k \text{ прелази ивицу } xy \text{ у својој путањи} \\ 0 & \text{у супротном} \end{cases} \tag{3}$$

Где је L_k укупна дужина путање мрава k, а Q је константа.

3 Имплементација

У овом поглављу описана је имплементација мрављег алгоритма у Ц програмском језику. Прва имплементација је секвенцијална и служи као полазна тачка за анализу перформанси. Након тога развијене су још две верзије алгоритма – једна која користи *OpenMP* библиотеку за паралелизацију на више процесорских језгара, и друга која користи *CUDA* технологију за извршавање на графичким процесорима. Циљ ових адаптација је смањење укупног времена извршавања и постизање ефикаснијег коришћења расположивих рачунарских ресурса.

3.1 Секвенцијална имплементација

Основна логика алгоритма имплементирана је кроз функцију `aco` која као параметар прима матрицу растојања између градова `distances`. Њену имплементацију могуће је видети у изворном коду 1.

```
1  void aco(int distances[NUM_CITIES][NUM_CITIES]) {
2    double pheromones[NUM_CITIES][NUM_CITIES];
3    initPheromones(pheromones);
4
5    int antPaths[NUM_ANTS][NUM_CITIES];
6    int antPathLengths[NUM_ANTS];
7    int visited[NUM_ANTS][NUM_CITIES];
8
9    int shortestPathLength = INT_MAX;
10   int shortestPath[NUM_CITIES];
11   int noImprovement = 0;
12
13   for (int iterNum = 0; iterNum < NUM_ITERATIONS;
     ↪  iterNum++) {
14     initAntStates(antPaths, visited);
15     for (int ant = 0; ant < NUM_ANTS; ant++) {
16       int pathLenght = 0;
17       for (int move = 1; move < NUM_CITIES; move++) {
18         int previousCity = antPaths[ant][move - 1];
19         double probabilities[NUM_CITIES] = {0.0};
20
21         setProbabilities(probabilities, visited[ant],
           ↪  pheromones, previousCity,
22                          distances);
```

```
23
24          int nextCity = decideNext(probabilities);
25          antPaths[ant][move] = nextCity;
26          visited[ant][nextCity] = 1;
27          pathLenght += distances[previousCity][nextCity];
28        }
29
30        pathLenght += distances[antPaths[ant][NUM_CITIES -
           ↪  1]][antPaths[ant][0]];
31        antPathLengths[ant] = pathLenght;
32
33        if (pathLenght < shortestPathLength) {
34          noImprovement = 0;
35          shortestPathLength = pathLenght;
36          for (int i = 0; i < NUM_CITIES; i++) {
37            shortestPath[i] = antPaths[ant][i];
38          }
39        }
40      }
41
42      if (noImprovement > MAX_NO_IMPROVEMENT) {
43        printf("Convergence on iter %d\n", iterNum);
44        break;
45      }
46
47      noImprovement++;
48
49      evaporatePheromones(pheromones);
50      addPheromones(pheromones, antPaths, antPathLengths);
51    }
52
53    printf("Shortest: %d\n", shortestPathLength);
54    printPath(shortestPath);
55  }
```

Изворни код 1: Главна функција секвенцијалне имплементације

У коду се може приметити да ће бити `NUM_ITERATIONS` генерација са по `NUM_ANTS` мрава. Функција `initAntStates` ће на почетку сваке генерације поставити по једног мрава у сваки од градова и на основу тога ће се генерисати њихове путање. За

```
1 void initAntStates(int antPaths[NUM_ANTS][NUM_CITIES],
2                     int visited[NUM_ANTS][NUM_CITIES]) {
3   for (int i = 0; i < NUM_ANTS; i++) {
4     for (int j = 0; j < NUM_CITIES; j++) {
5         antPaths[i][j] = j == 0 ? i : 0;
6         visited[i][j] = i == j;
7     }
8   }
9 }
```

Изворни код 2: Функција за иницијализацију мрава у почетне позиције

то се користе матрице `antPaths` која чува путању сваког мрава и `visited` која бележи посећене градове. Поступак постављања мрава може се видети у изворном коду 2.

У сваком од корака мрава потребно је прво израчунати вероватноће преласка из тренутног града у следећи. Ове вероватноће се чувају у матрици `probabilities`, чије вредности се рачунају у функцији `setProbabilities` на линији 21 у изворном коду 1. Код ове функције имплементира формулу 1 и приказан је у изворном коду 3

Следећи корак у алгоритму јесте одабир следећег града генерисањем псеудослучајне вредности. Овај део алгоритма реализован је у функцији `decideNext` и приказан у изворном коду 4.

Ови кораци се понављају све док сваки мрав не обиђе сваки град. Када су све путање мрава познате могуће је извршити испарење феромона помоћу функције `evaporatePheromones` (изворни код 5) и додавање феромона помоћу функције `addPheromones` (изворни код 6).

Цео поступак се понавља из генерације у генерацију, водећи рачуна о најбољем забележеном режултату (најкраћој путањи). Уколико алгоритам конвергира или прође максималан број генерација, проглашава се крај алгоритма и приказује се најкраћа путања.

3.2 *OpenMP* имплементација

3.2.1 Почетна запажања

Основна идеја паралелизације извршавања овог алгоритма заснива се на паралелном израчунавању путања за све мраве у оквиру исте генерације. Оваква стратегија је могућа јер путања сваког мрава зависи само од удаљености градова и фе-

```
void setProbabilities(double probabilities[NUM_CITIES], int visited[NUM_CITIES],
                      double pheromones[NUM_CITIES][NUM_CITIES],
                           int previousCity, int distances[NUM_CITIES][NUM_CITIES])
                           {
  for (int nextCity = 0; nextCity < NUM_CITIES; nextCity++) {
    if (!visited[nextCity]) {
      probabilities[nextCity] =
          pow(pheromones[previousCity][nextCity], ALPHA) *
          pow((1 / (double)distances[previousCity][nextCity]),
          BETA);
    }
  }

  double sumProbabilities = sumArray(probabilities, NUM_CITIES);

  for (int i = 0; i < NUM_CITIES; i++) {
    probabilities[i] /= sumProbabilities;
  }
}
```

Изворни код 3: Функција која рачуна вероватноће преласка у следећи град

```
int decideNext(double probabilities[NUM_CITIES]) {
  double decision = rand() / (double)RAND_MAX;

  double sum = 0;
  for (int i = 0; i < NUM_CITIES; i++) {
    sum += probabilities[i];
    if (sum >= decision) {
      return i;
    }
  }

  return NUM_CITIES - 1;
}
```

Изворни код 4: Функција која одлучује о следећој дестинацији мрава

```
void evaporatePheromones(double
 ↪  pheromones[NUM_CITIES][NUM_CITIES]) {
  for (int i = 0; i < NUM_CITIES; i++) {
    for (int j = 0; j < NUM_CITIES; j++) {
      pheromones[i][j] *= (1 - EVAPORATION_RATE);
    }
  }
}
```

Изворни код 5: Функција која рачуна испаравање феромона

```
1  void addPheromones(double
    ↪  pheromones[NUM_CITIES][NUM_CITIES],
2                     int antPaths[NUM_ANTS][NUM_CITIES],
3                     int antPathLengths[NUM_ANTS]) {
4    for (int ant = 0; ant < NUM_ANTS; ant++) {
5      for (int move = 0; move < NUM_CITIES - 1; move++) {
6        int source = antPaths[ant][move];
7        int dest = antPaths[ant][move + 1];
8        pheromones[source][dest] += Q /
          ↪  (double)antPathLengths[ant];
9        pheromones[dest][source] = pheromones[source][dest];
10     }
11     int source = antPaths[ant][NUM_CITIES - 1];
12     int dest = antPaths[ant][0];
13     pheromones[source][dest] += Q /
        ↪  (double)antPathLengths[ant];
14     pheromones[dest][source] = pheromones[source][dest];
15   }
16 }
```

Изворни код 6: Функција која додаје феромоне на основу дужине путања

ромона из претходне генерације. Управо ова зависност представља и ограничење – неопходно је ажурирати вредности феромона на крају сваке генерације мрава, па није могуће рачунати више од једне генерације паралелно. Анализом алгоритма приказаног у изворном коду 1, може се уочити да је могуће паралелизовати извршавање петље која се налази на линији 15, док петља на линији 13 мора остати секвенцијална.

3.2.2 Паралелно генерисање путања

Узимајући у обзир наведена ограничења, паралелна имплементација се своди на додавање `#pragma omp parallel for` директиве као што се може видети на линији 12 изворног кода 7.

```
1  void acoParallel(int distances[NUM_CITIES][NUM_CITIES]) {
2    double pheromones[NUM_CITIES][NUM_CITIES];
3    initPheromones(pheromones);
4
5    int shortestPathLength = INT_MAX;
6    int shortestPath[NUM_CITIES];
7    int noImprovement = 0;
8
9    for (int iterNum = 0; iterNum < NUM_ITERATIONS;
     ↪  iterNum++) {
10     double pheromoneUpdate[NUM_CITIES][NUM_CITIES] = {0.0};
11
12 #pragma omp parallel for reduction(+ : pheromoneUpdate)
13     for (int ant = 0; ant < NUM_ANTS; ant++) {
14       int path[NUM_CITIES];
15       int visited[NUM_CITIES] = {0};
16       int pathLength = 0;
17
18       path[0] = ant;
19       visited[ant] = 1;
20
21       for (int move = 1; move < NUM_CITIES; move++) {
22         int previousCity = path[move - 1];
23         double probabilities[NUM_CITIES] = {0.0};
24
25         setProbabilities(probabilities, visited,
           ↪  pheromones, previousCity,
```

```
                        distances);

        int nextCity = decideNext(probabilities);
        path[move] = nextCity;
        visited[nextCity] = 1;
        pathLength += distances[previousCity][nextCity];
      }

      pathLength += distances[path[NUM_CITIES -
       ↪  1]][path[0]];

      setPheromoneUpdate(pheromones, path, pathLength);

#pragma omp critical
      {
        if (pathLength < shortestPathLength) {
          noImprovement = 0;
          shortestPathLength = pathLength;
          for (int i = 0; i < NUM_CITIES; i++) {
            shortestPath[i] = path[i];
          }
        }
      }
    }

    if (noImprovement > MAX_NO_IMPROVEMENT)
      break;

    noImprovement++;

    evaporatePheromones(pheromones);
    addPheromoneUpdate(pheromones, pheromoneUpdate);
  }

  printf("Shortest: %d\n", shortestPathLength);
  printPath(shortestPath);
}
```

Изворни код 7: Адаптација главне функције помоћу *OpenMP* директива

3.2.3 Лажно дељење

Осим додавања директиве било је неопходно направити неколико измена како би се алгоритам додатно оптимизовао и прилагодио паралелном извршавању. Главна разлика ова два алгоритма огледа се у томе што матрица `antPaths` не постоји, већ је замењена појединачним низом `path` за сваког мрава, а по истом принципу матрица `visited` такође не постоји већ је замењена истоименим низом. Оваква оптимизација је битна јер би употреба матрица довела до појаве лажног дељења, и значајно нарушила перформансе програма. Наиме, првобитно ове матрице су биле подељене тако да сваки од мрава добије по један ред како би чувао своју путању и посећене градове, што би доводило до тога да различите нити, константо приступају и ажурирају релативно блиске меморијске локације. Због своје близине, постоји шанса да ове локације заврше у кеш линији више нити, што би довело до потребе да се ажурира кеш код свих других нити, иако оне никада нису имале потребу да притупају том делу меморије. Овај проблем је решен тако што сваки мрав (који ће увек припасти једној нити) чува путању и посећене градове одвојено и тиме избегава учестало поништавања кеша.

3.2.4 Редукција

Решавањем проблема лажног дељења наилази се на нови проблем. С обзиром да су путање сада приватне, и животни век им је дугачак колико и паралелни регион, поставља се питање како ажурирати феромоне, с обзиром да то мора бити урађено након што сви мрави заврше своје путање?

Један начин јесте да се направи помоћна матрица `pheromoneUpdate` која ће садржати вредности које треба додати на матрицу феромона, а ову матрицу попунити методом редукције. Начин на који је ово урађено видљив је на линији 12 изворног кода 7, а своди се на то да свака нит добија копију матрице `pheromoneUpdate` у коју уписује своје вредности. Затим се редукцијом све ове матрице спајају у једну која ће касније бити искоришћена за ажурирање вредности феромона у функцији `setPheromoneUpdate` на линији 55 изворног кода 7.

3.2.5 Најкраћа путања

По завршетку генерисања путање за сваког од мрава потребно је проверити да ли је пронађена нова најкраћа путања и уколико јесте сачувати је. У случају када би било потребно сачувати само дужину најкраће путање, ово би било могуће постићи редукцијом, али будући да је потребно сачувати и низ градова којима је мрав прошао, неопходно је направити комплексније решење. Путања и њена дужина чуваће се у дељеним променљивама `shortestPath` и `shortestPathLength`, а њихово попуњавање ће бити заштићено критичном секцијом (линија 38 изворног кода 7),

јер би у супротном истовремени проналазак краће путање више нити довео до трке до података.

3.3 *CUDA* имплементација

3.3.1 Мотивација

С обзиром на учесталу употребу матрица описаној у поглављу 3.1, може се донети закључак о великом потенцијалу за побољшање перформанси алгоритма употребом графичког копроцесора. Управо то нам омогућава *CUDA* технологија, која нам нуди интерфејс за употребу *Nvidia GPU (Graphical Processing Unit)* уређаја који располажу са великим бројем језгара и омогућавају масивну паралелизацију операција над матрицама.

3.3.2 Заузимање меморије

Први корак извршавања *CUDA* програма је заузимање неопходне меморије на уређају. На линијама 17 - 23 изворног кода 8 могуће је видети како се заузима меморија неопходна за чување путања мрава, као и матрица удаљености градова и матрица феромона. Будући да је матрица удаљености улазни параметар функције, њене вредности је неопходно копирати у заузету меморију на уређају што учињено у линији 25.

```
1  void aco(int distances[NUM_CITIES][NUM_CITIES]) {
2    int antPaths[NUM_ANTS][NUM_CITIES];
3    int antPathLengths[NUM_ANTS];
4
5    int shortestPathLength = INT_MAX;
6    int shortestPath[NUM_CITIES];
7    int noImprovement = 0;
8
9    int *antPaths_d, *visited_d, *antPathLengths_d;
10   int sizePerAnt = NUM_ANTS * NUM_CITIES * sizeof(int);
11
12   float *pheromones_d;
13   int *distances_d;
14   int sizePerCityInt = NUM_CITIES * NUM_CITIES *
      ↪  sizeof(int);
15   int sizePerCityFloat = NUM_CITIES * NUM_CITIES *
      ↪  sizeof(float);
```

```
16
17    cudaCheckError(cudaMalloc((void **)&antPaths_d,
      ↪  sizePerAnt));
18    cudaCheckError(cudaMalloc((void **)&visited_d,
      ↪  sizePerAnt));
19    cudaCheckError(
20        cudaMalloc((void **)&antPathLengths_d, NUM_ANTS *
          ↪  sizeof(int)));
21
22    cudaCheckError(cudaMalloc((void **)&pheromones_d,
      ↪  sizePerCityFloat));
23    cudaCheckError(cudaMalloc((void **)&distances_d,
      ↪  sizePerCityInt));
24
25    cudaCheckError(cudaMemcpy(distances_d, distances,
      ↪  sizePerCityInt,
26                              cudaMemcpyHostToDevice));
27
28    initPheromonesKernel<<<NUM_CITIES, 128>>>(pheromones_d);
29
30    for (int iterNum = 0; iterNum < NUM_ITERATIONS;
      ↪  iterNum++) {
31
32      initAntStatesKernel<<<NUM_ANTS, 128>>>(antPaths_d,
        ↪  visited_d);
33
34      antKernel<<<NUM_ANTS, 128>>>(antPaths_d, visited_d,
        ↪  pheromones_d,
35                                   distances_d, antPathLengths_d,
                                         ↪  iterNum);
36
37      cudaCheckError(
38          cudaMemcpy(antPaths, antPaths_d, sizePerAnt,
            ↪  cudaMemcpyDeviceToHost));
39      cudaCheckError(cudaMemcpy(antPathLengths,
        ↪  antPathLengths_d,
40                                NUM_ANTS * sizeof(int),
                                   ↪  cudaMemcpyDeviceToHost));
41
42      int bestAnt = -1;
```

```
43      for (int ant = 0; ant < NUM_ANTS; ant++) {
44        int pathLength = antPathLengths[ant];
45        if (pathLength < shortestPathLength) {
46          shortestPathLength = pathLength;
47          bestAnt = ant;
48          noImprovement = 0;
49        }
50      }
51      if (bestAnt != -1) {
52        for (int i = 0; i < NUM_CITIES; i++) {
53          shortestPath[i] = antPaths[bestAnt][i];
54        }
55      }
56
57      if (noImprovement > MAX_NO_IMPROVEMENT) {
58        printf("Convergence on iter %d\n", iterNum);
59        break;
60      }
61
62      noImprovement++;
63
64      evaporatePheromonesKernel<<<NUM_CITIES,
         ↪  128>>>(pheromones_d);
65      addPheromonesKernel<<<NUM_ANTS, 128>>>(pheromones_d,
         ↪  antPaths_d,
66                                             antPathLengths_d);
67    }
68
69    cudaFree(antPaths_d);
70    cudaFree(visited_d);
71    cudaFree(antPathLengths_d);
72
73    cudaFree(pheromones_d);
74    cudaFree(distances_d);
75
76    printf("Shortest: %d\n", shortestPathLength);
77    printPath(shortestPath);
78  }
```

Изворни код 8: Главна функција *CUDA* имплементације

```
__global__ void initPheromonesKernel(float *pheromones_d) {
  int i = blockIdx.x;
  int j = threadIdx.x;

  if (j < NUM_CITIES) {
    pheromones_d[i * NUM_CITIES + j] = 1.0;
  }
}
```

Изворни код 9: Иницијализација матрице феромона на графичком процесору

```
__global__ void initAntStatesKernel(int *antPaths_d, int
↪  *visited_d) {
  int ant = blockIdx.x;
  int i = threadIdx.x;

  if (i < NUM_ANTS) {
    antPaths_d[ant * NUM_CITIES + i] = i == 0 ? ant : 0;
    visited_d[ant * NUM_CITIES + i] = i == ant;
  }
}
```

Изворни код 10: Постављање мрава у почетне позиције помоћу графичког процесора

3.3.3 Постављање почетних стања

Пре почетка извршавања алгоритма, неопходно је поставити вредности у матрици феромона на почетне вредности (конкретно свака ивица графа ће имати почетну вредност 1). Ово је тривијална операција, која захтева да се свако поље у матрици додели једном од језгара графичког процесора и једноставно изврши додела вредности. Имплементација ове функције може се видети у изворном коду 9. Веома слична имплементација потребна је и за постављање мрава у почетне позиције на почетку сваке генерације, што је приказано у изворном коду 10.

3.3.4 Формирање путања

Као што је споменуто у поглављу 3.2.1, могуће је паралелно формирати путање за сваког мрава из исте генерације. Због тога је функција `antKernel` видљива на линији 34 изворног кода 8, покренута са по једним блоком нити за сваког мрава. Имплементација ове функције видљива је у изворном коду 11. Формирање путање је комплексно и састоји се од више корака који морају бити извршени секвенцијално. Упркос томе, могуће је значајно убрзати поједине послове, као што је рачунање вероватноћа преласка у следећи град. Након овог корака потребно је синхронизовати све нити како бисмо били сигурни да су све вероватноће израчунате, пре него што се изабере следећи град. Одабир града нема смисла извршавати на више од једне нити, па је тај задатак, заједно са потребом за синхронизацијом нити, представља уско грло алгоритма.

```
1  __global__ void antKernel(int *antPaths_d, int *visited_d,
   ↪  float *pheromones_d,
2                            int *distances_d, int
                              ↪  *antPathLengths_d,
3                            unsigned long long seed) {
4    int ant = blockIdx.x;
5    int pathLenght = 0;
6    curandState d_state;
7    curand_init(seed, ant, 0, &d_state);
8
9    for (int move = 1; move < NUM_CITIES; move++) {
10     int previousCity = antPaths_d[ant * NUM_CITIES + move -
        ↪  1];
11     __shared__ float probabilities[NUM_CITIES];
12
13     setProbabilities(probabilities, visited_d,
        ↪  pheromones_d, previousCity,
14                      distances_d, ant);
15
16     __syncthreads();
17
18     if (threadIdx.x == 0) {
19       int nextCity;
20       decideNext(probabilities, &d_state, &nextCity);
21       antPaths_d[ant * NUM_CITIES + move] = nextCity;
22       visited_d[ant * NUM_CITIES + nextCity] = 1;
```

```
23        pathLenght += distances_d[previousCity * NUM_CITIES +
           ↪  nextCity];
24      }
25
26      __syncthreads();
27    }
28
29    if (threadIdx.x == 0) {
30      pathLenght +=
31          distances_d[antPaths_d[ant * NUM_CITIES + NUM_CITIES
             ↪  - 1] * NUM_CITIES +
32                      antPaths_d[ant * NUM_CITIES]];
33      antPathLengths_d[ant] = pathLenght;
34    }
35  }
```

Изворни код 11: Формирање путања на графичком процесору

3.3.5 Проналажење најкраће путање

Након што се путање генеришу потребно је ажурирати најкраћу путању уколико је пронађена оптималнија вредност. Овај задатак није практично извршити употребом графичких језгара због великих шанси појаве трке за подацима. Из тог разлога на линијама 37-62 изворног кода 8, може се видети да је овај задатак извршен уз помоћ процесора.

3.3.6 Ажурирање феромона

Линије 64 и 65 изворног кода 8 приказују позиве функција задужених за ажурирање вредности феромона. Ове вредности се чувају у матрици, па је посупак њиховог ажурирања уз помоћ графичког процесора тривијалан. Вреди споменути да је да ажурирање вредности коришћена операција атомичног сабирања, јер постоји да више нити покуша истовремено да ажурира исто поље матрице. Имплементацију функције `addPheromonesKernel` могуће је видети у изворном коду 12.

```
__global__ void addPheromonesKernel(float *pheromones_d,
    int *antPaths_d,
                                    int *antPathLengths_d) {
  int ant = blockIdx.x;
  int move = threadIdx.x;

  __shared__ double pathLength;
  pathLength = (float)antPathLengths_d[ant];

  if (move < NUM_CITIES - 1) {
    int source = antPaths_d[ant * NUM_CITIES + move];
    int dest = antPaths_d[ant * NUM_CITIES + move + 1];
    float updateVal = Q / pathLength;
    atomicAdd(pheromones_d + source * NUM_CITIES + dest,
        updateVal);
    atomicAdd(pheromones_d + dest * NUM_CITIES + source,
        updateVal);
  } else if (move == NUM_CITIES - 1) {
    int source = antPaths_d[ant * NUM_CITIES + move];
    int dest = antPaths_d[ant * NUM_CITIES];
    float updateVal = Q / pathLength;
    atomicAdd(pheromones_d + source * NUM_CITIES + dest,
        updateVal);
    atomicAdd(pheromones_d + dest * NUM_CITIES + source,
        updateVal);
  }
}
```

Изворни код 12: Ажурирање вредности феромона помоћу графичког процесора

4 Резултати

Мрављи алгоритам не гарантује проналажење оптималне путање која обилази све дате градове тачно једном и враћа се у почетни град. У циљу евалуације тачности и перформанси, алгоритам је тестиран на скуповима од 20, 50, 80 и 100 градова. У наставку су приказани добијени резултати, укључујући дужине путања, као и поређење времена извршавања између све 3 имплементације.

С обзиром на то да је алгоритам није егзактне природе, резултати све 3 имплементације — секвенцијалне *OpenMP* и *CUDA* — садржаће мале варијације у погледу тачности решења, па се поређење већински односи на утицај паралелизације на перформансе извршавања.

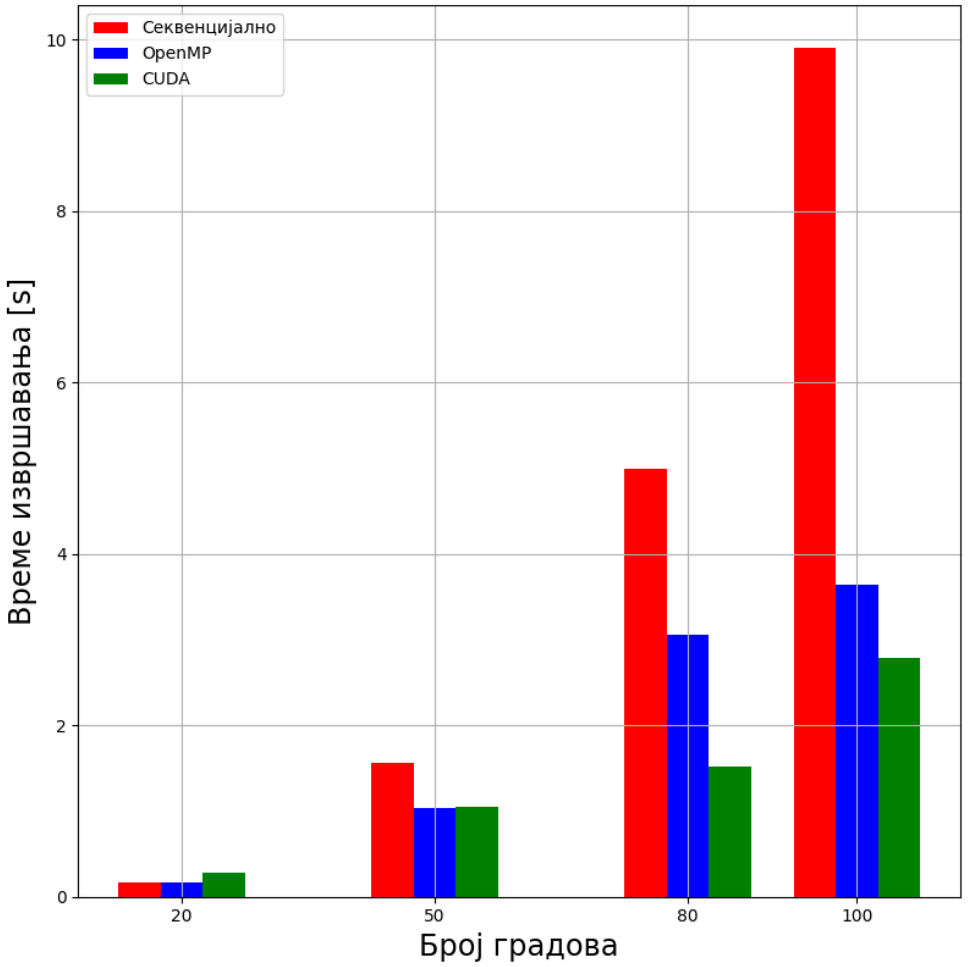
Подаци коришћени за тестирање алгоритма матрице растојања, са насумично генерисаним растојањима између сваког пара градова. Вредности растојања су генерисане у опсегу од 10 до 500.

4.1 Перформансе и решења

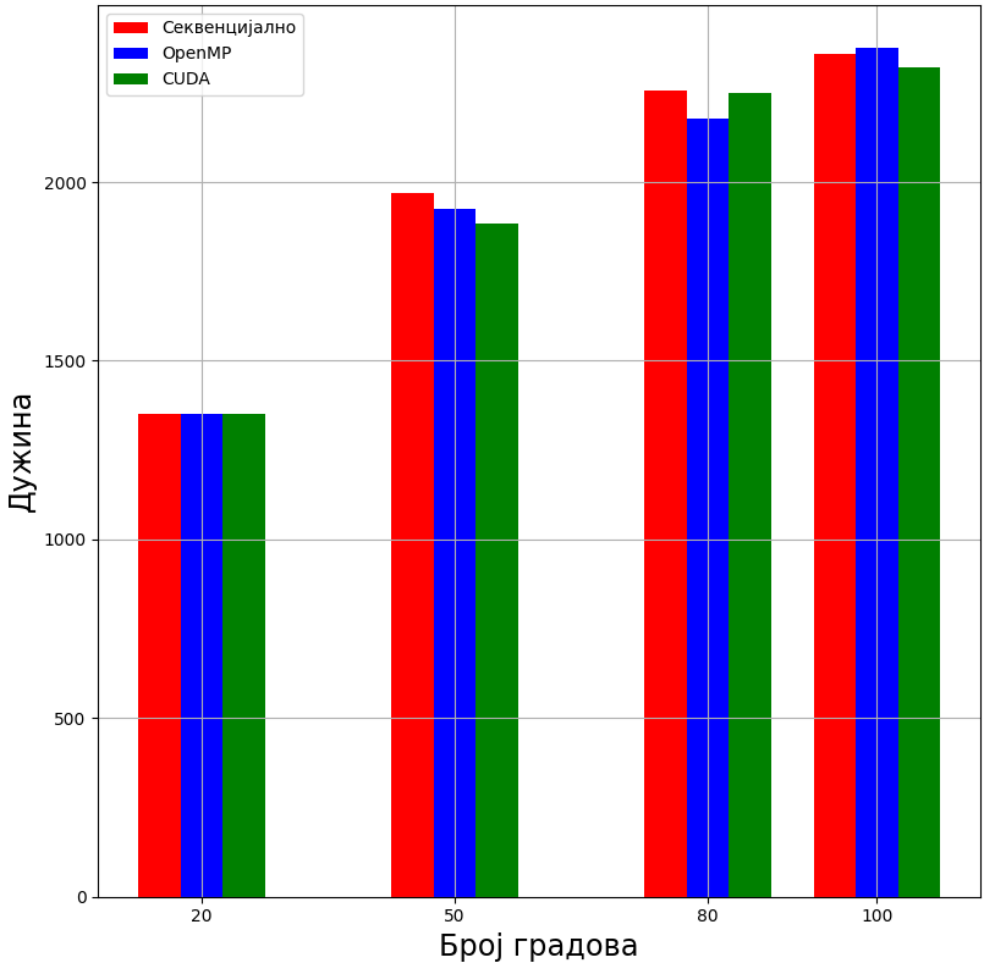
Табела 1 приказује времена извршавања и резултате за сваки од тестних скупова. Извршавање је обављено на процесору са 6 физичких и 12 логичких језгара, и *Nvidia GTX 1060 6GB* графичким процесором. Ради веће прегледности и лакше визуелне анализе, резултати су додатно представљени графички на сликама 1 и 2.

	Секвенцијална		*OpenMP*		*CUDA*	
20 градова	169ms	1350	165ms	1350	285ms	1350
50 градова	1.55s	1969	1.04s	1924	1.05s	1884
80 градова	4.99s	2255	3.05s	2177	1.51s	2251
100 градова	9.9s	2359	3.64s	2377	2.79s	2321

Табела 1: Времена извршавања секвенцијалног и паралелног алгоритма



Слика 1: Поређење времена извршавања секвенцијалне, *OpenMP* и *CUDA* имплементације



Слика 2: Поређење резултата извршавања секвенцијалне, *OpenMP* и *CUDA* имплементације

На основу времена извршавања може се приметити да за мали број градова (≤ 20 *CUDA* имплементација постиже нешто спорије перформансе због малих величниа матрица и времена потребног за заузимање меморије и копирање података са уређаја. Међутим, како се број градова повећава, време извршавања секвенцијалног алгоритма расте нагло, док *OpenMP* и *CUDA* показују израженије убрзање. Ово указује на то да се корист од паралелизације значајно испољава тек при већим и рачунски интензивнијим инстанцама проблема.

5 Закључак

На основу спроведене анализе и експеримената може се закључити да мрављи алгоритам, представља ефикасан метод за приближно решавање проблема трговачког путника и других комбинаторно сложених задатака. Имплементација у програмском језику Ц, уз коришћење *OpenMP* и *CUDA* технологија, омогућила је детаљно испитивање предности и изазова паралелног програмирања како на процесорима, тако и на графичким процесорима. Резултати показују да *OpenMP* имплементација постиже значајно убрзање на вишејезгарним *CPU* системима, док *CUDA* верзија додатно повећава перформансе код великих инстанци проблема, захваљујући масовној паралелизацији. Ово указује на велики потенцијал паралелних архитектура у примени мрављег алгоритма, посебно када се пажљиво управља ресурсима и оптимизује приступ меморији.

Библиографија

[1] Wikipedia. Ant colony optimization algorithms --- Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Ant%20colony%20optimization%20algorithms&oldid=1292514538, 2025. [Online; accessed 13-August-2025].

[2] Wikipedia. CUDA --- Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=CUDA&oldid=1305637637, 2025. [Online; accessed 13-August-2025].

[3] Wikipedia. OpenMP --- Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=OpenMP&oldid=1305617044, 2025. [Online; accessed 13-August-2025].

[4] Wikipedia. Travelling salesman problem --- Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Travelling%20salesman%20problem&oldid=1297255372, 2025. [Online; accessed 02--July-2025].