Przetwarzanie rozproszone i równoległe

Sprawozdanie z przygotowania projektu
Prowadzący: dr inż. Sławomir Bąk

Przygotował:
Krzysztof Kosman

Temat projektu

PVM - Wyszukiwanie najkrótszej trasy w grafie wykorzystując algorytm Dijkstry.
Repozytorium GIT: https://github.com/kkosman/pvm1-dijsktra

Przebieg pracy

1. Uruchomić testowy program PVM - hello world z zajęć.
   Dzięki temu zapoznałem się z ogólnymi zasadami działania środowiska PVM.

```c
#include "pvm3.h"
#include <stdio.h>

int main() {
        printf("Hello World\n");
        pvm_exit();
}
```

2. Stworzyć własny kod obsługujący algorytm Dijkstry dla małego grafu (2 wierzchołki) działający w głównym wątku.
   Poniższy algorytm wyszukuje długość najkrótszej ścieżki wg. algorytmu Dijkstry od wybranego wierzchołka początkowego (w naszym wypadku jest to zawsze 0) do wszystkich pozostałych.

```c
// A C++ program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph

#include <limits.h>
#include <stdio.h>
#include <stdbool.h>

// Number of vertices in the graph
#define V 9

// A utility function to find the vertex with minimum distance value, from
// the set of vertices not yet included in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
        // Initialize min value
        int min = INT_MAX, min_index;

        for (int v = 0; v < V; v++)
                if (sptSet[v] == false && dist[v] <= min)
                        min = dist[v], min_index = v;

        return min_index;
}

// A utility function to print the constructed distance array
void printSolution(int dist[])
{
        printf("Vertex \t\t Distance from Source\n");
        for (int i = 0; i < V; i++)
                printf("%d \t\t %d\n", i, dist[i]);
}

// Function that implements Dijkstra's single source shortest path algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
        int dist[V]; // The output array. dist[i] will hold the shortest
        // distance from src to i

        bool sptSet[V]; // sptSet[i] will be true if vertex i is included in shortest
        // path tree or shortest distance from src to i is finalized

        // Initialize all distances as INFINITE and stpSet[] as false
        for (int i = 0; i < V; i++)
                dist[i] = INT_MAX, sptSet[i] = false;

        // Distance of source vertex from itself is always 0
        dist[src] = 0;
```

```
        // Find shortest path for all vertices
        for (int count = 0; count < V - 1; count++) {
                // Pick the minimum distance vertex from the set of vertices not
                // yet processed. u is always equal to src in the first iteration.
                int u = minDistance(dist, sptSet);

                // Mark the picked vertex as processed
                sptSet[u] = true;

                // Update dist value of the adjacent vertices of the picked vertex.
                for (int v = 0; v < V; v++)

                        // Update dist[v] only if is not in sptSet, there is an edge from
                        // u to v, and total weight of path from src to v through u is
                        // smaller than current value of dist[v]
                        if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                                && dist[u] + graph[u][v] < dist[v])
                                dist[v] = dist[u] + graph[u][v];
        }

        // print the constructed distance array
        printSolution(dist);
}

// driver program to test above function
int main()
{
        /* Let us create the example graph discussed above */
        int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

        dijkstra(graph, 0);

        return 0;
}
```

3. Stworzenie prostego przykładu wykorzystującego mechanizmy komunikacji PVM Master:

```c
#include "pvm3.h"
#include <stdio.h>

#define SLAVENAME "slave"
#define SLAVENUM 4
#define NAMESIZE 64
#define MSG_MSTR 1
#define MSG_SLV 2

int main() {
        int mytid;
        int tids[SLAVENUM];
        char slave_name[NAMESIZE];
        int nproc, i, who;
        nproc = pvm_spawn(SLAVENAME, NULL, PvmTaskDefault, "", SLAVENUM, tids);
        mytid = pvm_mytid();
        printf("Master tid: %d \n", mytid);
        printf("debug: %d \n", nproc);

        for (i = 0 ; i < nproc ; i++) {
                pvm_initsend(PvmDataDefault);
                pvm_pkint(&mytid, 1, 1);
                pvm_pkint(&i, 1, 1);
                pvm_send(tids[i],MSG_MSTR);
        }
        for (i = 0 ; i < nproc ; i++) {
                pvm_recv(-1, MSG_SLV);
                pvm_upkint(&who, 1,1);
                pvm_upkstr(slave_name);
                printf("Master: proces %x jest na hoscie %s \n", who, slave_name);
        }

        pvm_exit();
}
```

Slave:

```
1
2   #include "pvm3.h"
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <unistd.h>
6
7   #define SLAVENAME "slave"
8   #define SLAVENUM 4
9   #define NAMESIZE 64
10  #define MSG_MSTR 1
11  #define MSG_SLV 2
12
13  int main() {
14          int ptid, my2, mytid;
15          char *str = malloc(NAMESIZE);
16          int i = pvm_recv(-1, MSG_MSTR);
17          if( i < 0 ) {
18                  perror("Nieoczekiwany blad");
19                  exit(0);
20          }
21          pvm_upkint(&ptid, 1, 1);
22          pvm_upkint(&my2, 1, 1);
23          printf("Slave: otrzymalem %x %d\n", ptid, my2);
24          mytid = pvm_mytid();
25          pvm_initsend(PvmDataDefault);
26          pvm_pkint(&mytid, 1,1);
27          gethostname(str, NAMESIZE);
28          str[NAMESIZE - 1] = 0;
29          pvm_pkstr(str);
30          pvm_send(ptid, MSG_SLV);
31          pvm_exit();
32  }
```

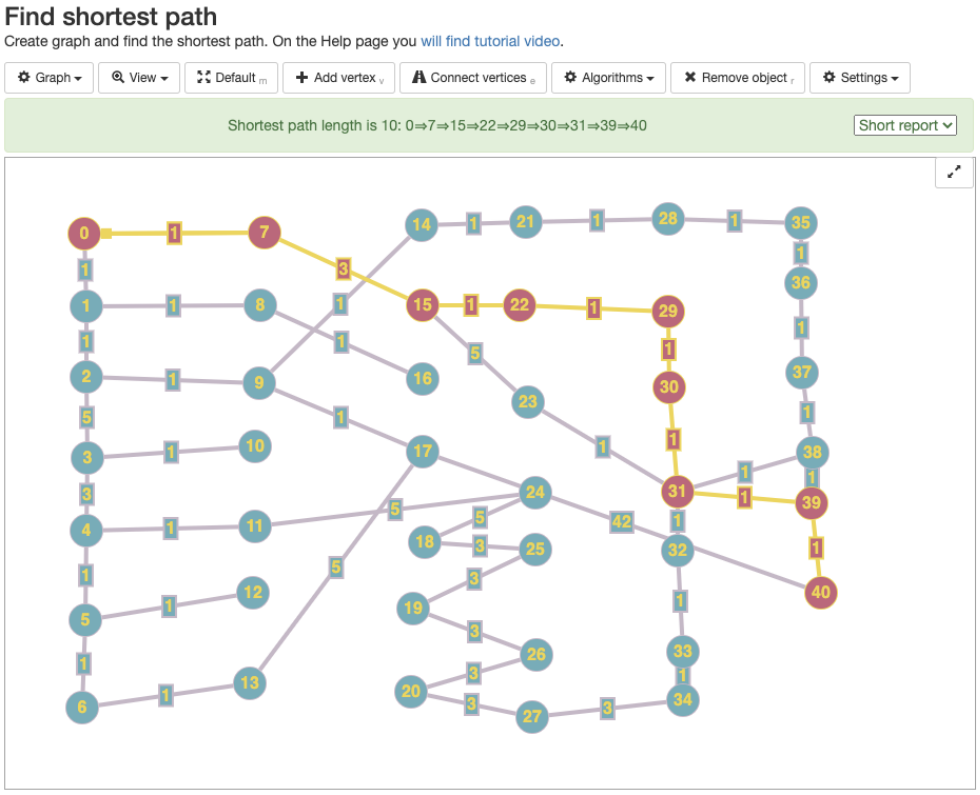
4. Przenieść część obliczeń algorytmu do programu Slave
   Zostało to wykonane w rewizji:
   https://github.com/kkosman/pvm1-dijsktra/tree/cf6085d44ab62a7acbaefe6fee86c27df52e7553
5. Przetestować działanie dla małego grafu (2 wierzchołki)

6. Zbudować duży testowy graf (41 wierzchołków) przy pomocy narzędzia graphonline.ru
7. Utworzyć testowe środowisko z wykorzystaniem VirtualBox i Ubuntu składające się z 3 maszyn.

```
test@master:~/projects/pvm1-dijsktra$ export PVM_ROOT="/usr/lib/pvm3"
test@master:~/projects/pvm1-dijsktra$ export PVM_ARCH=`$PVM_ROOT/lib/pvmgetarch`
test@master:~/projects/pvm1-dijsktra$ export PVM_PATH=$HOME/pvm3/bin/$PVM_ARCH
test@master:~/projects/pvm1-dijsktra$ export PVM_SRC=$HOME/pvm3/src
test@master:~/projects/pvm1-dijsktra$ export PATH=$PATH:$PVM_ROOT/bin:$PVM_ROOT/
lib:$PVM_HOME
test@master:~/projects/pvm1-dijsktra$ export XPVM_ROOT=/usr/bin/xpvm
test@master:~/projects/pvm1-dijsktra$ export PVM_RSH=`which ssh`
test@master:~/projects/pvm1-dijsktra$ printf "%s\n%s\n" conf quit|${PVM_ROOT}/li
b/pvm machines.conf
pvm> conf
3 hosts, 1 data format
                     HOST     DTID     ARCH   SPEED       DSIG
                   master    40000  LINUX64    1000 0x00408c41
                    test1    80000  LINUX64    1000 0x00408c41
                    test2    c0000  LINUX64    1000 0x00408c41
pvm> quit

Console: exit handler called
pvmd still running.
```

8. Uruchomić projekt testując na 3 hostach. Wyniki testów umieszczam poniżej.
9. Sprawdzenie poprawności algorytmu.
   Algorytm oblicza dla każdego wierzchołka grafu najkrótszą ścieżkę od wybranego wierzchołka. W moim wypadku jest to wierzchołek 0. Sprawdzam, przykład: Najkrótsza droga od węzła 0 do węzła 40 = 10. Weryfikuję to z narzędziem Graphonline. Wynik jest zgodny.



Napotkane problemy

1. Stworzenie dużego testowego, poprawnego grafu. Na szczęście z pomocą przyszedł serwis https://graphonline.ru/en/ który udostępnia świetne narzędzie online za pomocą którego mogłem wykonać graf nieukierunkowany z wagami na potrzeby zadania o 41 wierzchołkach. Pozwolił mi też wygenerować dla niego jego programową reprezentację w postaci adjacency matrix.
2. Trudnym okazało się również połączenie kilku instancji Ubuntu w sieć przez VirtualBox. Na szczęście udało się ten problem rozwiązać korzystając z Google.
3. Problem z dodaniem hostów do PVM. Po skonfigurowaniu sieci i sprawdzeniu komunikacji przez ssh, okazało się że PVM wyłączał się przy każdej próbie dodania nowego hosta. Pomoc znalazłem na Stack Overflow: https://stackoverflow.com/questions/2253354/pvm-terminates-after-adding-host

Efekt

Testy przeprowadzałem na komputerze MacBook Pro, środowisko PVM uruchomiłem na Ubuntu 20.04 przez VirtualBox.

Tabela wyników pomiaru czasów przetwarzania:

Rozmiar grafu (wierzchołki)	**Ilość hostów**	**Moc procesora**	**Czas przetwarzania**
2	1	100% x 4 core	0.000475
9	1	100% x 4 core	0.000792
9	2	100% x 4 core	0.000789
41	1	100% x 4 core	0.001872
41	2	100% x 4 core	0.002290
41	3	100% x 4 core	0.002221
41	1	10% x 1 core	0.001049
41	3	10% x 1 core	0.002090

Screeny z działania:

1. Działanie na 1 hoście dla grafu o 9 wierzchołkach

```
pvm> spawn -> master
spawn -> master
[1]
1 successful
t40002
pvm> [1:t40003] EOF
[1:t40004] EOF
[1:t40005] EOF
[1:t40006] EOF
[1:t40007] EOF
[1:t40008] EOF
[1:t40009] EOF
[1:t40002] Vertex                Distance from Source
[1:t40002] 0             0
[1:t40002] 1             4
[1:t40002] 2             12
[1:t40002] 3             19
[1:t40002] 4             21
[1:t40002] 5             11
[1:t40002] 6             9
[1:t40002] 7             8
[1:t40002] 8             14
[1:t40002] Execution time 0.000792 seconds.
[1:t40002] EOF
[1:t4000a] EOF
```

2. 2 hosty, 9 wierzchołków

```
spawn -> master
[4]
1 successful
t40013
pvm> [4:t80012] EOF
[4:t80013] EOF
[4:t80014] EOF
[4:t80015] EOF
[4:t80016] EOF
[4:t40014] EOF
[4:t40015] EOF
[4:t40013] Vertex                Distance from Source
[4:t40013] 0             0
[4:t40013] 1             4
[4:t40013] 2             12
[4:t40013] 3             19
[4:t40013] 4             21
[4:t40013] 5             11
[4:t40013] 6             9
[4:t40013] 7             8
[4:t40013] 8             14
[4:t40013] Execution time 0.000789 seconds.
[4:t40013] EOF
[4:t40016] EOF
```

3. Działanie na 1 hoście dla grafu o 2 wierzchołkach

```
pvm> spawn -> master
spawn -> master
[1]
1 successful
t40002
pvm> [1:t40002] Vertex           Distance from Source
[1:t40002] 0             0
[1:t40002] 1             1
[1:t40002] Execution time 0.000475 seconds.
[1:t40003] EOF
[1:t40002] EOF
```

4. Działanie na 1 hoście, 41 wierzchołków

```
[1:t40002] 5            11
[1:t40002] 6            10
[1:t40002] 7            1
[1:t40002] 8            2
[1:t40002] 9            3
[1:t40002] 10           8
[1:t40002] 11           11
[1:t40002] 12           12
[1:t40002] 13           9
[1:t40002] 14           4
[1:t40002] 15           4
[1:t40002] 16           3
[1:t40002] 17           4
[1:t40002] 18           21
[1:t40002] 19           23
[1:t40002] 20           17
[1:t40002] 21           5
[1:t40002] 22           5
[1:t40002] 23           9
[1:t40002] 24           16
[1:t40002] 25           24
[1:t40002] 26           20
[1:t40002] 27           14
[1:t40002] 28           6
[1:t40002] 29           6
[1:t40002] 30           7
[1:t40002] 31           8
[1:t40002] 32           9
[1:t40002] 33           10
[1:t40002] 34           11
[1:t40002] 35           7
[1:t40002] 36           8
[1:t40002] 37           9
[1:t40002] 38           9
[1:t40002] 39           9
[1:t40002] 40           10
[1:t40002] Execution time 0.001872 seconds.
[1:t40002] EOF
```

5. 2 hosty, 41 wierzcholkow

```
[1:t80001] 9            3
[1:t80001] 10           8
[1:t80001] 11           11
[1:t80001] 12           12
[1:t80001] 13           9
[1:t80001] 14           4
[1:t80001] 15           4
[1:t80001] 16           3
[1:t80001] 17           4
[1:t80001] 18           21
[1:t80001] 19           23
[1:t80001] 20           17
[1:t80001] 21           5
[1:t80001] 22           5
[1:t80001] 23           9
[1:t80001] 24           16
[1:t80001] 25           24
[1:t80001] 26           20
[1:t80001] 27           14
[1:t80001] 28           6
[1:t80001] 29           6
[1:t80001] 30           7
[1:t80001] 31           8
[1:t80001] 32           9
[1:t80001] 33           10
[1:t80001] 34           11
[1:t80001] 35           7
[1:t80001] 36           8
[1:t80001] 37           9
[1:t80001] 38           9
[1:t80001] 39           9
[1:t80001] 40           10
[1:t80001] Execution time 0.002290 seconds.
[1:t80014] EOF
[1:t80001] EOF
```

6. 3 hosty 41 wierzchołków

```
[1:t80001] 21            5
[1:t80001] 22            5
[1:t80001] 23            9
[1:t80001] 24            16
[1:t80001] 25            24
[1:t80001] 26            20
[1:t80001] 27            14
[1:t80001] 28            6
[1:t80001] 29            6
[1:t80001] 30            7
[1:t80001] 31            8
[1:t80001] 32            9
[1:t80001] 33            10
[1:t80001] 34            11
[1:t80001] 35            7
[1:t80001] 36            8
[1:t80001] 37            9
[1:t80001] 38            9
[1:t80001] 39            9
[1:t80001] 40            10
[1:t80001] Execution time 0.002221 seconds.
[1:t8000d] EOF
[1:t80001] EOF
```

7. cap 10%, 1 host, 41 wierzchołków

```
[1:t40002] 24            16
[1:t40002] 25            24
[1:t40002] 26            20
[1:t40002] 27            14
[1:t40002] 28            6
[1:t40002] 29            6
[1:t40002] 30            7
[1:t40002] 31            8
[1:t40002] 32            9
[1:t40002] 33            10
[1:t40002] 34            11
[1:t40002] 35            7
[1:t40002] 36            8
[1:t40002] 37            9
[1:t40002] 38            9
[1:t40002] 39            9
[1:t40002] 40            10
[1:t40002] Execution time 0.001049 seconds.
[1:t4002a] EOF
[1:t40002] EOF
[1:t40026] EOF
[1:t4001e] EOF
[1:t40022] EOF
```

8. cap 10%, 3 host, 41 wierzchołków

```
[1:t80001] 21            5
[1:t80001] 22            5
[1:t80001] 23            9
[1:t80001] 24            16
[1:t80001] 25            24
[1:t80001] 26            20
[1:t80001] 27            14
[1:t80001] 28            6
[1:t80001] 29            6
[1:t80001] 30            7
[1:t80001] 31            8
[1:t80001] 32            9
[1:t80001] 33            10
[1:t80001] 34            11
[1:t80001] 35            7
[1:t80001] 36            8
[1:t80001] 37            9
[1:t80001] 38            9
[1:t80001] 39            9
[1:t80001] 40            10
[1:t80001] Execution time 0.002090 seconds.
[1:t8000d] EOF
[1:t80001] EOF
```

Wnioski

1. PVM pozwala rozproszyć obliczenia na wiele maszyn, ukrywa szczegóły komunikacji oraz ściąga z barki programisty konieczność dbania o różnice w reprezentacji danych dla różnych środowisk programistycznych. Dołączanie nowych maszyn jest bardzo proste i nie wymaga zmiany w samym kodzie algorytmu.
2. W przypadku prostego algorytmu i szybkich procesów komunikacja pomiędzy hostami potrafi zniwelować zyski z rozproszenia obliczeń. Tak się stało w moim przykładzie. Wykonanie programu na jednym hoście zajmowało mniej czasu, niż na 2 lub 3. Co wskazuje na to, że czas potrzebny na komunikację był wyższy niż czas potrzebny na wykonanie obliczeń.
3. Nie udało mi się zaobserwować zysków z rozproszenia obliczeń dla algorytmu Dijkstry. Przypuszczam, że może to oznaczać, że algorytm jest na tyle optymalny, że współczesne maszyny bez problemu sobie z nim radzą w pojedynkę lub zastosowałem zbyt mały graf do testów. Jednak tworzenie większych grafów jest już bardzo mocno utrudnione.