Obliczanie liczby π metodą Monte-Carlo

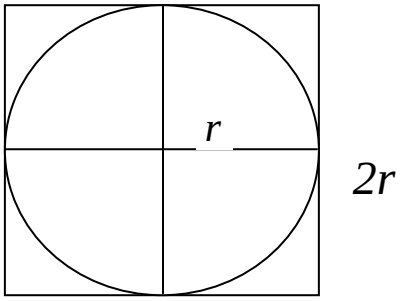
Zakres ćwiczenia

Ćwiczenie obecne podzielone jest na dwie części. Celem pierwszej części ćwiczenia podniesienie umiejętności w zakresie stosowania dotąd nabytej wiedzy. Tak więc w tej części ćwiczenia nie zostaną wprowadzone żadne nowe funkcje biblioteki PVM. Zostanie przedstawiona metoda wyliczania liczby π, która stosunkowo dobrze nadaje się do zrównoleglenia. W drugiej części zostanie przedstawiony sposób użycia wielu buforów komunikacyjnych w środowisku PVM.

Opis metody Monte-Carlo obliczania liczby π

Liczbę π można obliczać na wiele różnych sposobów. Sposób, który podamy obecnie nie jest bynajmniej najlepszym z nich – głównym motywem wybrania właśnie jego jest tzw. cel dydaktyczny. Metoda Monte-Carlo, z którą zaraz się zapoznasz, jest dość łatwa do zrozumienia oraz dobrze daje się ją zrównoleglić, co czyni ją idealną do wszelkiego rodzaju kursów programowania w środowisku rozproszonym i tradycyjnie pojawia się w wielu ćwiczeniach uczących programowania w Ada95, PVM czy też MPI.

Obejrzyj teraz rysunek poniżej. Przedstawia on koło o promieniu *r* wpisane w kwadrat o boku *2r*.



Rysunek 1 Koło wpisane w kwadrat

Gdybym zadał ci teraz pytanie, ile wynosi pole przedstawionego wyżej kwadratu, na pewno odpowiedziałbyś bez wahania $4r^2$. Pole koła zaś wynosi πr^2. Oznacza to, że stosunek pola koła do pola kwadratu wynosi:

$$\frac{P_{kola}}{P_{kwadrat}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4} \qquad 6.1$$

Z tego z kolei wynika, że mając obliczone wcześniej w jakiś sposób pole kwadratu i pole koła wpisanego w ten kwadrat, możemy łatwo obliczyć π:

$$\pi = 4\frac{P_{kola}}{P_{kwadrat}} \qquad 6.2$$

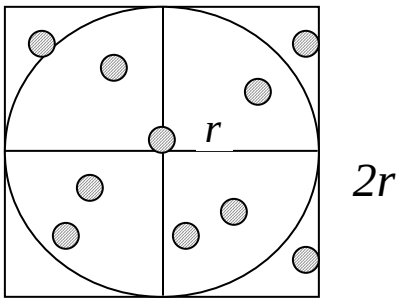
Ta konstatacja być może wywołała na twojej twarzy uśmiech. W końcu, żeby obliczyć pole koła, musimy znać π, mógłbyś powiedzieć. Owszem, to prawda. Ale istota metody Monte-Carlo polega na tym, że możemy zastosować powyższą równość bez obliczania pola koła.

Wyobraź sobie teraz, że rzucamy ziarenkami piasku w narysowany kwadrat z zakreślonym w środku kołem. Zgodzisz się chyba z stwierdzeniem, że jeśli będziemy rzucali dostatecznie długo, w końcu ziarnka piasku pokryją cały kwadrat, a stosunek liczby ziarenek piasku w środku narysowanego koła w stosunku do wszystkich ziarenek piasku w całym kwadracie będzie równy mniej więcej stosunkowi pola koła do pola kwadratu. Bardziej formalnie możemy ten wniosek ubrać w słowa w następujący sposób:

Jeżeli będziemy losować punkty o współrzędnych od *–2r* do *2r*, to stosunek liczby punktów zawierających się w kole o środku w punkcie <0,0> i promieniu *r* do wszystkich wylosowanych punktów będzie dążył w nieskończoności (z pewnym prawdopodobieństwem) do stosunku tego pola koła do koła kwadratu o boku *2r*.

Co więcej, stosunek ten będzie identyczny również do ćwiartki koła. Jeżeli pole koła podzielimy na cztery i tak samo podzielimy pole kwadratu, to ich stosunek będzie wciąż taki sam. Oznacza to, że wystarczy, jeżeli będziemy losowali punkty o współrzędnych od 0 do *r*.

Cała metoda sprowadza się więc do tego, by losować punkty, sprawdzać, czy mieszczą się w kole, i następnie podstawiać liczby wylosowanych punktów do wzoru 6.2. Losując **odpowiednio dużo** punktów, powinniśmy otrzymać z pewnym prawdopodobieństwem rozsądne przybliżenie liczby π.



Rysunek 2 Przykład losowania punktów w metodzie Monte-Carlo

Powyżej znajduje się prosty przykład. Wylosowano 10 punktów, z czego 3 znalazły się poza kołem. Widać z tego, że przybliżenie π wyniosło 4*7/10=2.8, co jest wprawdzie dość odległe od prawdy, ale pokazuje ogólną ideę – im więcej będzie punktów, tym przybliżenie π bliższe faktycznej wartości 3,14159265358979323846264338327 95....

Zadanie do samodzielnego wykonania

Twoim zadaniem obecnie jest napisanie programu który będzie obliczał liczbę π wyżej opisaną metodą Monte-Carlo. Aby ułatwić zadanie, poniżej zostanie umieszczony szkielet programu. Sam natomiast powinieneś zdecydować w jaki sposób podzielić obliczenia i jak zaimplementować algorytm. Oczywiście rozwiązanie tego zadania odnajdziesz między materiałami kursu, ale postaraj się najpierw samemu napisać program.

Jak zwykle, program będzie składał się z dwóch części: *mastera* oraz procesów *slave*. Proces *master* oczywiście może brać udział w obliczeniach na równych prawach jak reszta procesów. Jego zadaniem będzie pobranie argumentów, rozesłanie ich między procesy *slave* i zebranie wyników.

Program master.c

```
1. int main(int argc, char **argv)
2. {
3.         int R, points, slaves;
4.         int nproc;
5.         int *tids;
6.         if (argc < 4)
7.               return -1;
8.         R = atoi(argv[1]);
9.         points = atoi(argv[2]);
10.        slaves = atoi(argv[3]);
11.        tids = (int *)malloc(slaves*sizeof(int));
12.        nproc=pvm_spawn("slave",0,0,".",slaves,tids);

13.        for (i=0;i<slaves;i++)
14.        {
15.              pvm_initsend( PvmDataRaw );
16.              pvm_pkint( &R, 1,1);
17.              pvm_pkint( &points, 1,1);
18.              pvm_send( tids[i], 1 );
19.        }
```

W powyższym szkielecie programu najpierw pobierane są argumenty (linijki 8-10). Argument pierwszy to będzie *r* koła. Argument drugi to parametr określający liczbę punktów losowany przez procesy *slave*. Argument trzeci określa liczbę procesów typu *slave*. Jeżeli nie znasz funkcji `atoi`, zapoznaj się z jej opisem za pomocą polecenia `man atoi`. W linijce 10 następuje przydział pamięci dla tablicy `tids`, która będzie zawierać identyfikatory uruchomionych procesów typu *slave*.

Następnie w pętli do każdego procesu wysyłane są otrzymane parametry – wielkość koła i liczba punktów.

Liczba π powinna być wyliczana na przykład za pomocą następującego kodu: `4.0 * ((double)inside_circle)/(double)total`, gdzie `inside_circle` oznacza liczbę wylosowanych punktów mieszczących się w środku koła, a `total` liczbę wszystkich wylosowanych punktów.

Program slave.c powinieneś móc napisać samemu. Poniżej znajdziesz tylko sposób losowania punktów o współrzędnych od 0 do *r* (losujemy więc punkty tylko w ćwiartce kwadratu i koła):

```
1. srand( pvm_mytid() );
2. x = rand()%R;
3. y = rand()%R;
```

Aby zwiększyć losowość, generator liczb pseudolosowych jest inicjowany w linijce 1-szej identyfikatorem zadania PVM; zapewnia to fakt zainicjowania inną wartością generatora dla każdego zadania PVM.

Do sprawdzenia, czy wylosowany punkt leży w obrębie koła należy po prostu sprawdzać, czy odległość wylosowanego punktu od środka układ współrzędnych jest mniejsza do *r*, a odległość tę można wyliczyć korzystając z wzoru Pitagorasa.

Przykładowy wynik uruchomienia programu może wyglądać na przykład tak:

```
pvm> spawn -> master 200 10 20
[2]
1 successful
t40068
pvm> [2:t40069] EOF
 [2:t4006a] EOF
 [2:t4006b] EOF
 [2:t4006c] EOF
                        ...
[2:t40068] Master: Biezace PI to: 2.400000
[2:t40068] Master: Biezace PI to: 3.000000
[2:t40068] Master: Biezace PI to: 3.200000
[2:t40068] Master: Biezace PI to: 3.300000
[2:t40068] Master: Biezace PI to: 3.360000
[2:t40068] Master: Biezace PI to: 3.333333
[2:t40068] Master: Biezace PI to: 3.257143
[2:t40068] Master: Biezace PI to: 3.250000
[2:t40068] Master: Biezace PI to: 3.155556
[2:t40068] Master: Biezace PI to: 3.120000
[2:t40068] Master: Biezace PI to: 3.127273
[2:t40068] Master: Biezace PI to: 3.133333
[2:t40068] Master: Biezace PI to: 3.076923
[2:t40068] Master: Biezace PI to: 3.114286
[2:t40068] Master: Biezace PI to: 3.146667
[2:t40068] Master: Biezace PI to: 3.075000
[2:t40068] Master: Biezace PI to: 3.082353
[2:t40068] Master: Biezace PI to: 3.111111
[2:t40068] Master: Biezace PI to: 3.115789
[2:t40068] Master: Biezace PI to: 3.140000
[2:t40068] Master: Odebralem wiadomosc od wszystkich
[2:t40068] EOF
[2:t4007a] EOF
[2] finished
```

Oczywiście, metoda Monte-Carlo jest *losowa*, co oznacza, że można otrzymać wyniki raz lepsze, raz gorsze w zależności od uruchomienia a powyższe wartości można uznać za wyjątkowo dobre, biorąc pod uwagę podane parametry.

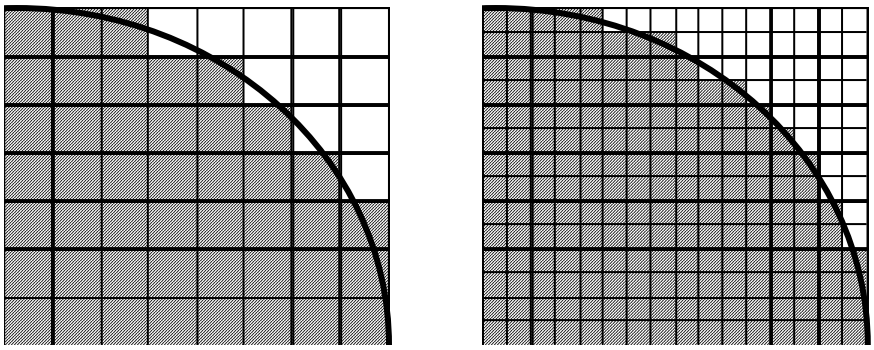
Wskazówki do wykonania ćwiczenia

Aby otrzymać dobre rozwiązanie, należy zwrócić uwagę na kilka kwestii.

Zasadą, której należy się trzymać zawsze w pisaniu programów rozproszonych, jest minimalizacja komunikacji. Im mniej procesy się komunikują i im więcej wykonują obliczeń lokalnie, tym większe przyśpieszenie. W przypadku wielu programów pisanych na ćwiczeniach stosunek obliczeń lokalnych do komunikacji jest tak niski, że praktycznie opóźnienia komunikacyjne i tak niweczą wszelkie pozytywne skutki rozproszenia aplikacji – niemniej jednak należy o tej zasadzie pamiętać.

Należy tak skonstruować program, by minimalizować liczbę błędów numerycznych. Błędy numeryczne mogą być tutaj wprowadzane przy operacjach dzielenia i (ewentualnie) obliczania pierwiastków (podczas sprawdzania, czy punkt mieści się w kole). Należy więc wybrać takie rozwiązanie, w którym tych operacji będzie jak najmniej. W przypadku naszego problemu, rozwiązanie dobre to takie, w którym nie będzie w ogóle operacji na pierwiastkach.

Im większe *r*, tym większa rozdzielczość losowania i większa szansa na dobre przybliżenie liczby π. Wynika to z tego, że operując na współrzędnych-liczbach całkowitych w gruncie rzeczy nie badamy prawdziwego koła, a jedynie pewne jego przybliżenie. Problem ten ilustruje rysunek poniżej.



Rysunek 3 Przybliżenia koła o promieniu *r*=8 oraz *r*=16 dla całkowitych współrzędnych

Na rysunku narysowano wycinek koła o promieniu *r* równym 8. Widać, że jeżeli losujemy tylko współrzędne całkowite, w gruncie rzeczy otrzymamy tylko pewne przybliżenie koła (obszar zakreskowany). Przybliżenie to będzie tym bardziej „koliste", im większe *r*,

Im więcej punktów, tym lepsze przybliżenie liczby π - jest to prawdą, ale tylko do pewnego stopnia. Jeżeli punktów jest za dużo w stosunku do promienia *r*, to przestajemy się zbliżać do właściwej wartości 3.14159... a rozwiązanie może się zacząć pogarszać. Wynika to z tego, że zaczynamy w końcu ponownie losować te same punkty. Jeżeli wybierzemy *r* równe 100, to nie ma sensu losować więcej niż 10000 punktów.

Wykorzystanie wielu buforów komunikacyjnych

Omówimy teraz funkcje, które można wykorzystać w przypadku, gdy użytkownik chce zarządzać wieloma buforami wewnątrz pojedynczej aplikacji. W przypadku PVM w każdej chwili istnieje jeden aktywny bufor nadawczy i jeden aktywny odbiorczy (tworzony automatycznie po wywołaniu funkcji odbierającej komunikaty, na przykład `pvm_recv`). W zupełności to wystarcza w większości zastosowań i można używać PVM-a nie będąc nawet świadom możliwości używania większej liczby buforów. W przypadku zajścia jednak takiej potrzeby, użytkownik może jednak stworzyć wiele buforów i później przełączać się pomiędzy nimi.

Przedstawiony poniżej program pokazuje sposób wykorzystania funkcji służących do zarządzania wieloma buforami. Składać się on będzie z dwóch plików, jednego o nazwie `master.c` zawierającego kod dla *mastera* oraz pliku `slave.c` z kodem dla procesów typu *slave*.

`program master.c`

```
1.   #include "def.h"
2.   #include <stdio.h>
3.   int main(int argc, char **argv)
4.   {
5.      int tids[SLAVENUM];
6.      int i=10;
7.      int buf1, buf2;
8.      pvm_spawn(SLAVENAME,0,0,".",SLAVENUM,tids);
9.      buf1 = pvm_mkbuf( PvmDataDefault );
10.     pvm_setsbuf( buf1 );
11.     pvm_pkint(&i,1,1);
12.     buf2 = pvm_mkbuf( PvmDataDefault );
13.     pvm_setsbuf( buf2 );
14.     i=20;
15.     pvm_pkint(&i,1,1);
```

Pierwsze osiem linijek nie powinny już wymagać wyjaśnienia. Proces *master* tworzy procesy potomne, a następnie (linijki 9 oraz 12) za pomocą funkcji `pvm_mkbuf` (o argumencie o znaczeniu identycznym jak dla wcześniej poznanej funkcji `pvm_initsend`) tworzy dwa bufory, których identyfikatory są zapamiętane w zmiennych `buf1` oraz `buf2`. W pierwszym z nich proces umieszcza liczbę 10, a w drugim liczbę 20 (linijki 11 oraz 15).

Należy zwrócić uwagę, że samo utworzenia bufora nie oznacza automatycznie, że od razu można w nim umieszczać dane za pomocą funkcji `pvm_pk*`. Najpierw bufor ten musi zostać jeszcze oznaczony jako aktywny (funkcja `pvm_setsbuf` w linijce 10 i 13)

```
16.      for (i=0;i<SLAVENUM;i++)
17.      {
18.            pvm_setsbuf( buf1 );
19.            pvm_send(tids[i],MSG_MSTR);
20.            pvm_setsbuf( buf2 );
21.            pvm_send(tids[i],MSG_SLV);
22.      }
23.      pvm_freebuf(buf1);
24.      pvm_freebuf(buf2);
25.      pvm_exit();
26.      }
```

Następnie do każdego procesu potomnego wysyłane są komunikaty, na zmianę, z bufora o identyfikatorze `buf1` oraz `buf2`. Przełączanie między buforami następuje za pomocą funkcji `pvm_setsbuf` (linijki 18 oraz 20). Wreszcie bufory są zwalniane za pomocą funkcji `pvm_freebuf` (linijki 23-24) – operacja ta nie jest potrzebna dla bufora utworzonego za pomocą funkcji `pvm_initsend`.

`Program slave.c`

```
1. #include "def.h"
2. #include <stdio.h>

3. int main()
4. {
5.        int i;
6.        int tab[5];

7.        pvm_recv(-1,MSG_SLV);
8.        pvm_upkint( &i, 1, 1);
9.        printf("Slave: %d a potem ",i);
10.       pvm_recv(-1,MSG_MSTR);
11.       pvm_upkint( &i, 1, 1);
12.       printf("%d \n ",i);
13.       pvm_exit();
14. }
```

Program *slave* odbiera dwie wiadomości za pomocą blokującej funkcji `pvm_recv` (przy okazji demonstrując możliwość zmiany kolejności odbierania komunikatów) w linijkach 7 oraz 10, wypakowuje z nich liczby (linijki 8 oraz 11) i wypisuje je na standardowym wyjściu.

Należy teraz skompilować oba programy i umieścić pliki wykonywalne w odpowiednim katalogu:

```
gcc master.c –o master –lpvm3

gcc slave.c –o slave –lpvm3

cp master slave $PVM_HOME
```

Wynik uruchomienia programu za pomocą komendy konsoli PVM `spawn -> master` (dla 4 procesów typu *slave*) może wyglądać tak:

```
pvm> spawn -> master
[1]
1 successful
t8000b
pvm> [1:t4000f] Slave: 10 a potem 20
[1:t4000f]  lave: 10 a potem 20
[1:t4000f] EOF
[1:t8000c] Slave: 10 a potem 20
[1:t8000b] EOF
[1:t8000d] Slave: 10 a potem 20
[1:t8000c]
[1:t8000c] EOF
[1:t8000d]  lave: 10 a potem 20
[1:t8000d] EOF
[1:t40010] Slave: 10 a potem 20
[1:t40010]
[1:t40010] EOF
[1] finished
```

Wynik ten demonstruje zarazem możliwe dziwaczne zachowanie w przypadku użycia jako argumentu dla funkcji `printf` łańcucha nie zakończonego znakiem końca linii, o ile przed pojawieniem się tego znaku pojawi się jakaś funkcja komunikacyjna PVM.

Poznane funkcje można wykorzystać także do przesyłania dalej otrzymanych wiadomości, bez potrzeby ich ponownego przepakowywania:

```
bufid = recv( -1, MSG_MSTR);
stary_bufor = setsbuf( bufid );
pvm_send( id_procesu, MSG_MSTR);
pvm_freebuf( stary_bufor);
```

Poznane funkcje biblioteki PVM

`int bufid = pvm_mkbuf (int encoding)`

Funkcja ta tworzy nowy pusty bufor wysyłający i ustawia dla niego metodę konwersji. Zwracany jest identyfikator bufora.

`int info = freebuf(int bufid)`

Funkcja ta zwalnia bufor o identyfikatorze bufid. Powinna być wywołana po tym jak komunikat został wysłany i nie jest już więcej potrzebny.

`int bufid = pvm_getsbuf(void)`

`int bufid = pvm_getrbuf(void)`

Funkcje te zwracają identyfikator aktywnego bufora nadawczego lub odbiorczego.

`int oldbuf = pvm_setsbuf(int bufid)`

`int oldbuf = pvm_setrbuf(int bufid)`

Funkcje te ustawiają aktywny bufor nadawczy bądź odbiorczy, zapamiętują stan poprzedniego bufora i zwracają jego identyfikator. Jeżeli argumentem funkcji jest 0 to zapamiętywany jest poprzedni bufor i aplikacja nie ma aktywnego bufora.

Podsumowanie

W trakcie tych ćwiczeń wykonałeś samodzielnie program obliczający liczbę π metodą Monte-Carlo. Zaznajomiłeś się w ten sposób z jednym z wielu zastosowań metody Monte-Carlo oraz podniosłeś swoją sprawność w używaniu środowiska PVM. Dodatkowo dowiedziałeś się, w jaki sposób używać wielu buforów komunikacyjnych.

Co powinieneś wiedzieć:

- Co to jest metoda Monte-Carlo obliczania liczby π
- W jaki sposób używać wielu buforów komunikacyjnych (`pvm_setsebuf`, `pvm_mkbuf`)