Zadanie 2

Zadanie to jest w swoim charakterze podobne do drugiego zadania z zeszłego roku. W tym roku będziemy pastwić się nad algorytmem, który dla podanego n oblicza kolejne, począwszy od $n+1$, cyfry po przecinku *szesnastkowego* rozwinięcia liczby π [1]. Algorytm ten bazuje na następującym rozwinięciu liczby π w szereg

$$\pi = \sum_{i=0}^{\infty} \frac{1}{16^i} \left(\frac{4}{8i+1} - \frac{2}{8i+4} - \frac{1}{8i+5} - \frac{1}{8i+6} \right).$$

Liczba dokładnie obliczonych cyfr zależy od ilości uwzględnionych składników. Prezentowany algorytm nie jest asymptotycznie najszybszym znanym dla problemu obliczania n początkowych cyfr rozwinięcia liczby π, ale jest bardzo prosty w implementacji i do obliczenia cyfr o numerach $n+1, n+2, \ldots$ nie wymaga znajomości cyfr poprzednich, co czyni go łatwym do zrównoleglenia na wielu procesorach.

Niech $frac(x)$ oznacza część ułamkową liczby x, czyli $frac(x) = x - \lfloor x \rfloor$. Dla danego n obliczamy $y = frac(16^n \cdot \pi)$. Wtedy kolejne cyfry po przecinku o numerach $1, 2, 3, \ldots$ rozwinięcia szesnastkowego liczby y są odpowiednio cyframi po przecinku o numerach $n+1, n+2, n+3, \ldots$ rozwinięcia szesnastkowego liczby π. Korzystamy z następującej tożsamości

$$y = frac(4S_1 - 2S_4 - S_5 - S_6),$$

gdzie

$$S_k = \sum_{i=0}^{n-1} \frac{16^{n-i} \bmod (8i+k)}{8i+k} + \sum_{i=n}^{\infty} \frac{16^{n-i}}{8i+k}.$$

Przy czym sumowanie drugiego z powyższych szeregów kończymy, gdy składniki stają się mniejsze niż zadany z góry ε.

Implementacja powyższego algorytmu wykonana w języku Fortran z użyciem arytmetyki zmiennopozycyjnej podwójnej precyzji dostępna jest na stronie internetowej [2]. Na tej stronie można też znaleźć m.in. pracę [1]. Program ten daje poprawne wyniki dla n do około 11860000 i wyznacza dobrze przynajmniej osiem cyfr rozwinięcia. Implementacja ta nie wydaje się jednak być dość optymalna i tu zaczyna się zadanie.

Należy wykonać efektywną implementację wyżej opisanego algorytmu w wersji jednoprocesorowej (zrównoleglenie na wiele procesorów może być tematem innego zadania). Można użyć arytmetyki zmiennopozycyjnej pojedynczej lub podwójnej precyzji lub arytmetyki stałopozycyjnej pojedynczej precyzji albo zaimplementować arytmetykę stałopozycyjną podwójnej lub wielokrotnej precyzji. Postać algorytmu wydaje się być idealna do zastosowania SSE2. Rozwiązaniem powinna być implementacja funkcji, którą można wywołać z poziomu języka C. Dopuszczalne są dwa prototypy tej funkcji:

```
double pids(int);
unsigned pids(int);
```

W obu przypadkach jedynym argumentem tej funkcji jest n. W pierwszym przypadku funkcja powinna zwracać obliczoną wartość y, a w drugim przypadku powinna zwracać 8 początkowych cyfr po przecinku szesnastkowego rozwinięcia liczby y, czyli początkowe 32 bity po przecinku dwójkowego rozwinięcia liczby y. Dla $n \leqslant 11860000$ funkcja powinna produkować przynajmniej 8 dobrych cyfr rozwinięcia szesnastkowego. Funkcja może używać co najwyżej polilogarytmicznej wielkości pamięci.

Prawdopodobnie niezłym sposobem podejścia do tego zadania będzie wykonanie najpierw dobrej implementacji w C, a potem przepisanie części lub całości w asemblerze lub użycie funkcji wbudowanych. Można używać funkcji z biblioteki matematycznej, czyli tej, którą dolinkowuje się za pomocą opcji `-lm`.

Ocena zadania będzie polegała na sprawdzeniu poprawności produkowanych wyników i zmierzeniu przyspieszenia (*improvement*) uzyskiwanego w stosunku do implementacji w Fortranie. Liczba punktów za zadanie będzie wyliczana jako średnia z kilku testów dla $n = 999999$, 1000000, 1000001, ... za pomocą formuły

$$2 \cdot (improvement - 1)$$

i zaokrąglana do 0,1. Dodatkowo będzie przyznawany 1 punkt, jeśli funkcja będzie działać poprawnie (dobrych co najmniej osiem cyfr rozwinięcia) dla $n \geqslant 16000000$. Student może wybrać komputer, na którym będzie wykonywany test programu: orangexx, whitexx, itd., students, własny laptop lub inny udostępniony na czas testu prowadzącemu zajęcia.

Do zadania dołączone są następujące pliki:

- `pi.f` – oryginalna implementacja w Fortranie;
- `pitest.c` – kod testujący;
- `pidsd.asm` – implementacja funkcji `double pids(int)` w asemblerze, a tak naprawdę wywołanie w tym celu oryginalnej implementacji;
- `pidsu.c` – implementacja funkcji `unsigned pids(int)` w C, a tak naprawdę wywołanie w tym celu oryginalnej implementacji;
- `makefile` – plik umożliwiający skompilowanie powyższych plików.

W wyniku kompilacji powstają dwa pliki wykonywalne:

- `pitestd` – używa funkcji `double pids(int)`,
- `pitestu` – używa funkcji `unsigned pids(int)`.

Programy `pitestd` i `pitestu` wywołuje się, podając jako argumenty dwie testowane wartości n.

Jako rozwiązanie zadania należy oddać pliki źródłowe w C, asemblerze, itp. oraz plik makefile, który umożliwi skompilowanie tych plików i ich zlinkowanie z plikami `pitest.c` i `pi.f` służącymi do testowania. Pliki `pitest.c` i `pi.f` należy kompilować z opcjami `-Wall -O3`. Niedopuszczalne jest rozwiązanie *offline*, czyli polegające na wcześniejszym stablicowaniu początkowych kilkunastu milionów cyfr rozwinięcia i przedstawieniu tylko samych wyników obliczeń. Ocenie podlega program, za pomocą którego wykonano obliczenia.

Pisząc w C, warto wypróbować różnych kombinacji opcji optymalizacji: `-O3`, `-march=pentium3`, `-march=pentium4`, `-msse2`, `-mfpmath=see`, `-fomit-frame-pointer`, itp.

Literatura

[1] D. Bailey, P. Borwein, S. Plouffe, On the rapid computation of various polylogarithmic constants, *Mathematics of Computation*, **66** (1997), 903–913.

[2] P. Borwein, *Home Page*, http://www.cecm.sfu.ca/personal/pborwein/.