WYŻSZA SZKOŁA HANDLOWA W RADOMIU

RADOM ACADEMY OF ECONOMICS

Wyższa Szkoła Handlowa w Radomiu

Programowanie równoległe
Laboratorium 1

Radom 2020/2021

1. Cel zadania

Celem laboratorium jest zapoznanie się z podstawowymi zaletami oraz wadami przetwarzania równoległego i/lub współbieżnego.

2. Potrzebne narzędzia.

- kompilator C++ minimum standardu 11 (optymalnie 14)
- biblioteki zrównoleglające zadania (np. OpenMPI, OpenMP)
- minimum dwuwątkowy procesor (optymalnie minimum dwurdzeniowy)

Opcjonalnie – TYLKO JEŻELI komputer posiada więcej niż dwa rdzenie:
- dowolne system wirtualizacji maszyn (VirtualBox, VMware, Hyper-V)
- dowolny system operacyjny wyposażony w kompilator z wymagań podstawowych

3. Informacje wstępne

Wraz z rozwojem informatyzacji rosną wymagania co do wydajności urządzeń elektronicznych. Wbrew pozorom wydajność jest wymagana nie tylko w sferze obliczeń i kalkulacji naukowo-matematycznych – w takim samym stopniu potrzebują tego aplikacje multimedialne i użytkowe. W przypadku aplikacji użytkowych większa moc potrzebna jest, gdy np. pracujemy przy obróbce zdjęć i/lub grafice wektorowej. W przypadku multimediów przetwarzanie wielowątkowe wykorzystuje się przy kompresji audio, wideo czy w grach komputerowych.

Sam proces przetwarzania współbieżnego i/lub równoległego występuje w naszym codziennym życiu. Nasze codzienne obowiązki powodują, że jednocześnie musimy zająć się kilkoma zadaniami naraz. Z doświadczenia możemy też wywnioskować, że nie jesteśmy w stanie realnie wykonywać dwóch rzeczy w pełni jednocześnie. Jedno z zagadnień zawsze wykonujemy „w przerwie” innego, bądź skupiamy swoją uwagę na drugim zadaniu gdy kontynuacja pierwszego zadania z przyczyn od nas niezależnych jest niemożliwa. Przykładem takiego działania współbieżnego może być np. szykowanie posiłku, podczas którego to procesu musimy poczekać na zagotowanie się wody. W zależności od złożoności naszego posiłku możemy w czasie oczekiwania na wodę zająć się przygotowaniem pozostałych, niezależnych składników, takich jak np. pieczywo.

Przykładem procesu równoległego może być zaś praca na budowie. Poszczególne procesy mogą występować równolegle, jak kopanie fundamentów i mieszanie betonu. Jednak co jakiś czas prace muszą być synchronizowane (by wylać fundament musimy mieć zakończone prace kopania).

Identycznie sprawa wygląda w przypadku elektroniki. Nie zawsze możliwe jest wykonanie prac w pełni niezależnie od siebie. Niekiedy szereg operacji, ze względu ograniczeń aplikacji i/lub elektroniki, będą musiały wykonywać się w ramach pojedynczego procesu (współbieżnie). Wtedy wykonywanie kilku operacji w ramach aplikacji (np. obsługa menu programu, aktualizacja danych, wyświetlenie wyników) będzie musiało wykonywać się w szczelinach czasowych lub wedle priorytetu danego elementu wobec pozostałych elementów (np. aktualizacja danych będzie najważniejsza, następnie obsługa przerwań interfejsu, na końcu zaś wyświetlanie wyników). W przypadku zaś operacji równoległych (przeliczania części większego zadania) prędzej czy później będzie od nas wymagać synchronizacji prac, czyli poczekania aż wszystkie elementy zadania zostaną ukończone (bądź będziemy oczekiwać na wyniki jednej części zadania by rozpocząć kolejne).

Jak można się domyślić, jednoczesne wykonywanie większej liczby zadań i/lub wątków to skomplikowany proces. Pisanie rozwiązania tego problemu mogłoby wiązać się z dużymi nakładami programistycznymi, co niekoniecznie mogłoby zakończyć się wykonaniem efektywnego przetwarzania wielowątkowego i/lub wielozadaniowego. Dlatego też najczęściej wykorzystuje się gotowe, dojrzałe rozwiązania pozwalające na pozyskanie jak najlepszych wyników przy znacznie

mniejszym zaangażowaniu programistycznym. Dodatkowym atutem gotowych rozwiązań jest możliwość wykorzystania ich w grupie wieloosobowej dzięki ujednoliceniu kodu.

Ćwiczenie będzie wykonywane na jednym, bazowym kodzie przemnażania dwóch macierzy MxN, gdzie dla uproszczenia M=N. Same wyniki nie są istotne – istotny jest czas wykonania obliczeń. Kod dostępny w dodatku do ćwiczenia.

4. Przebieg.

Pierwszym etapem ćwiczenia powinno być zapoznanie się z zasobami urządzenia, na którym będziemy wykonywać obliczenia. W tym wypadku dobrze jest przestawić sprzęt – model procesora i jego wersję.

Następnie należy skompilować i wykonać kod. Trzeba notować czas wykonania dla poszczególnych wielkości macierzy, zaczynając do rozmiaru 10. Testy należy zakończyć w chwili, gdy czas oczekiwania na wynik przekroczy 5 minut, jednak za ostateczny rozmiar macierzy wykorzystywany w ćwiczeniu należy uznać 10000 (nawet gdyby nadal przemnażanie wykonywało się poniżej założonego czasu). Można testowy rozmiar dowolnie modyfikować (np. sprawdzić czas dla wykonywania macierzy o rozmiarze 125x125 zamiast ze stu przeskakiwać na rozmiar 1000).

INFORMACJA: Czas można notować automatycznie wykorzystując np. zapis do pliku tekstowego.

Znając już możliwości wykonywania kodu sekwencyjnie, należy przetestować wbudowane rozwiązanie do zrównoleglania zadań. W tym celu trzeba skorzystać z funkcji thread: http://www.cplusplus.com/reference/thread/thread/. Wykorzystując te same wielkości macierzy co poprzednio należy zanotować szybkość wykonania zadania.

Kolejną częścią zadania powinno być sprawdzenie działania biblioteki OpenMP. Sposób użycia przedstawiany był na wykładzie; same przykłady (wraz z ćwiczeniami) można znaleźć również pod tym adresem: https://www.openmp.org/resources/tutorials-articles/.

Na koniec do zadania należy zastosować rozwiązanie OpenMPI. Sposób instalacji i użycia znajduje się pod tym adresem: https://mpitutorial.com/tutorials/mpi-hello-world/. Proszę zwrócić uwagę, że NIE MA gotowej wersji OpenMPI dla Windows – nikt się tym rozwiązaniem dla tego systemu nie zajmuje. Pozostaje kompilacja i ręczna instalacji bibliotek w systemie lub skorzystanie z Cygwin/maszyny wirtualnej Linux (lub instalacja systemu Linux na maszynie fizycznej).

UWAGA: Każdy przebieg należy wykonać dla maksymalnej ilości wątków dla swojej maszyny, dla połowy maksymalnej ilości wątków oraz dla ¼ sumarycznej ilości wątków. Przykładowo jeżeli mamy do dyspozycji 6 rdzeni 2 wątkowych (łącznie 12 wątków) to wykonujemy ćwiczenie dla 12 wątków, 6 oraz 3 wątków. Jeżeli mamy do dyspozycji 6 rdzeni 1 wątkowych – odpowiednio 6, 3 i 2 wątki aplikacji (ilość wątków zaokrąglamy w górę!).

Proszę pamiętać, że ćwiczenie znacznie zyska na czytelności, jeżeli mamy do dyspozycji wielordzeniową jednostkę, zaś samo ćwiczenie będzie wykonywane na maszynie wirtualnej. Wtedy mamy możliwość „fizycznej” modyfikacji ilości rdzeni i wątków, przez co wyniki będą bardziej dokładne. Ponadto można sprawdzić czy wątki wykonywane w środowisku wirtualnym będą się wykonywać szybciej, wolniej bądź w porównywalnym czasie.

5. Zakończenie/uwagi do sprawozdania

W sprawozdaniu, prócz samych wyników działania kodu (oraz jego szybkości wykonania) należy wskazać z jaką złożonością mamy do czynienia w załączonym kodzie. Ponadto, dla lepszej czytelności wyników należy zastanowić się nad wykonaniem wykresów czasowych dla poszczególnych rozwiązań/ilości wątków wykonujących zadanie. Czy zaproponowany kod wymaga

od nas synchronizacji wątków? Jeżeli tak – w którym momencie? Jeżeli nie – czy ma to znaczący wpływ na osiągnięte wyniki?

Dodatkowym atutem będzie zaproponowanie poprawek w kodzie zwiększających jego efektywność/uniwersalność.

DODATEK

```
#include <iostream>
#include <chrono>

using namespace std;

int main()
{
    const unsigned int m=10;
    const unsigned int n=10;

    srand(static_cast<unsigned int>(static_cast< std::chrono::duration<double>
>(std::chrono::high_resolution_clock::now().time_since_epoch()).count()));

    double **matrixa;
    double **matrixb;
    double **matrixc;

    matrixa = new double*[m];
    matrixb = new double*[m];
    matrixc = new double*[m];

    unsigned int max = static_cast<unsigned int>(1u<<31);

    for (unsigned int i=0;i<m;i++)
        matrixa[i]=new double[n];

    for (unsigned int i=0;i<m;i++)
        matrixb[i]=new double[n];

    for (unsigned int i=0;i<m;i++)
        matrixc[i]=new double[n];

    for (unsigned int i=0;i<m;i++)
        for (unsigned int j=0;j<n;j++)
            matrixa[i]
[j]=static_cast<double>(static_cast<double>(rand())/max*10);

    for (unsigned int i=0;i<m;i++)
        for (unsigned int j=0;j<n;j++)
            matrixb[i]
[j]=static_cast<double>(static_cast<double>(rand())/max*10);

    auto start=std::chrono::high_resolution_clock::now();

    for (unsigned int i=0;i<m;i++)
        for (unsigned int j=0;j<n;j++)
           for (unsigned int k=0;k<m;k++)
               for (unsigned int l=0;l<m;l++)
                   matrixc[i][j]+=matrixa[k][l]*matrixb[l][k];

    auto stop=std::chrono::high_resolution_clock::now();
```

```
    std::chrono::duration<double> time_diff=stop-start;

    cout << "Czas wykonania programu " << time_diff.count()<< " sekund." <<
endl;

    for (unsigned int i=0;i<m;i++)
        delete [] matrixa[i];

    for (unsigned int i=0;i<m;i++)
        delete [] matrixb[i];

    for (unsigned int i=0;i<m;i++)
        delete [] matrixc[i];

    delete [] matrixa;
    delete [] matrixb;
    delete [] matrixc;

    return 0;
}
```