Mnożenie przez macierz jako funkcja

Wstęp

Na tym laboratorium nie poznamy *sensum stricte* nowych metod numerycznych. Zamiast tego przyjrzymy się, jak w optymalny sposób zaimplementować metody, które już znamy. Wykonując poprzednie laboratorium powinniśmy byli zauważyć, że najbardziej wydajną metodą iteracyjną (przynajmniej z tych rozważanych) rozwiązania układu algebraicznego wynikającego z MES jest metoda gradientów sprzężonych z preconditionerem Jacobiego. Wobec tego skupimy się tylko na niej, ale warto zaznaczyć, że prezentowane poniżej rozwiązania z powodzeniem można stosować również dla pozostałych metod iteracyjnych.

Macierz $N \times N$ typu `double` zajmuje w pamięci $8N^2$ byte'ów, także np. mając 16GB pamięci RAM możemy zaalokować macierz reprezentującą układ równań o ok. 45tys. niewiadomych (nawet mniej, gdy uwzględnimy pamięć zarezerwowaną na inne zmienne, system operacyjny itd). Implementacja z poprzednich zajęć jest wobec tego zupełnie nieadekwatna do „prawdziwych" problemów inżynierskich, które potrafią mieć nawet dziesiątki milionów stopni swobody. Szczęśliwie okazuje się, że trzymanie w pamięci całego A nie jest potrzebne. Nietrudno zauważyć, że w metodzie gradientów sprzężonych nie używamy elementów macierzy, lecz tylko możliwości mnożenia przez nią. Innymi słowy, nie musimy wiedzieć jak wygląda A, wystarczy że dla danego wektora x potrafimy policzyć Ax. Dziś wykorzystamy to spostrzeżenie, aby przyspieszyć program i drastycznie zredukować ilość wykorzystywanej przez niego pamięci.

Dodatkowo, chętni mogą dowiedzieć się, jak wykorzystać współczesne wielordzeniowe procesory do przyspieszenia obliczeń.

Przygotowanie

Aby nie pomylić się w następnych krokach, należy najpierw zrefaktoryzować (tzn. przeorganizować i „posprzątać") kod. W tym celu upewnij się, że Twoja implementacja spełnia poniższe wymagania:

- Implementacja metody CG jest wydzielona do osobnej funkcji (dla ustalenia uwagi w tej instrukcji nazwiemy ją `Solve`)
- `Solve` przyjmuje następujące argumenty: rozmiar układu równań `N` (typ całkowity), macierz `A` (typ `double **`), wektor `b` (typ `double *`), wektor przybliżenia początkowego `x0` (typ `double *`) oraz wektor, do którego wpisane zostać ma rozwiązanie `x` (typ `double*`).
- `Solve` nie zwraca wartości lub ewentualnie zwraca kod błędu (typ `int`).
- Wewnątrz `Solve` jasno zdefiniowana jest maksymalna liczba iteracji `max_it` (typ `const int`) oraz poziom zbieżności `eps` (typ `const double`). Opcjonalnie możesz te wielkości przekazywać jako dodatkowe argumenty.
- Poszczególne elementy danej iteracji, np. liczenie residuum, mnożenie macierz-wektor, mnożenie skalarne wektorów itp. są wydzielone do osobnych funkcji.
- Bezpośrednio w `Solve` nie występują odniesienia do poszczególnych elementów `A` (są one przeniesione do odpowiednich funkcji, z których `Solve` korzysta)
- Zmienna `fix` jest globalna.

Element po elemencie

Skopiuj funkcję do mnożenia macierz-wektor i nazwij kopię `SMult`. W funkcji `SMult` będziemy chcieli napisać funkcję mnożącą przez globalną macierz sztywności S nie używając samej tej macierzy. Chcemy wykonać operację $r = Sx$, tzn: $r_i = \sum_j S_{ij} x_j$.

Chcielibyśmy aby funkcja `Mult_A` wykonywała mnożenie pewnego wektora $\mathbf{v}$ przez globalną macierz sztywności nie używając samej macierzy $\mathbf{A}$. To znaczy, że chcemy wykonać operację $\mathbf{t} = \mathbf{A}\mathbf{v}$ czyli:

$$t_i = \sum_{j=1}^{N} A_{ij} v_j, \ (i = 1 \dots N)$$

Przypomnijmy, że globalną macierz sztywności $\mathbf{A}$ tworzy się przez sumowanie elementów macierzy lokalnych. Zastanówmy się więc co się dzieje z wynikiem mnożenia $\mathbf{t}$ jeśli do macierzy $\mathbf{A}$ dodamy coś.

Analogicznie jeśli dodamy do elementu S_{ij} liczbę w, to tak jak byśmy dodali do elementu r_i liczbę $w \cdot x_j$. Jako, że macierz S konstruujemy właśnie przez dodawanie do kolejnych jej elementów, możemy całość mnożenia przez nią zapisać w powyższej postaci.

Zadanie

Przekopiuj fragment kodu funkcji `main` odpowiedzialny za konstrukcję macierzy S. Następnie, każde wystąpienie

```
S[i,j] += cos;
```

zamień na:

```
r[i] += cos * x[j];
```

Jeżeli planujesz realizować część instrukcji dot. równoległości, przerób kod tak, aby aktualizował tablicę `r` blokami, tzn. wykonywał mnożenie *lokalnej* macierzy sztywności przez odpowiedni „wycinek" wektora `x`, wynik wpisywał do bufora (typ `double[8]`), a bufor dodawał do tablicy `r` dopiero po wykonaniu tego mnożenia. Pomoże to uniknąć nadmiernej synchronizacji wątków. Nawiasem mówiąc, takie rozwiązanie może się okazać nieco bardziej wydajne również dla sekwencyjnej (nierównoległej) implementacji. Wynika to z faktu, że korzystając z alokowanego na stosie bufora dokonujemy mniejszej liczby dostępów (operacji `+=`) do tablicy `r`, która zaalokowana jest na stercie.

Zadanie

Co z częścią, która zamieniała wybrane wiersze na wiersze macierzy diagonalnej? Jeśli w macierzy S i-ty wiersz zamienimy na same zera i 1 na przekątnej, to tak jak byśmy postawili $r_i = x_i$. Zamień pętlę wycinającą $i$ty wiersz, na `r[i]=x[i]`

Zadanie 8

Jeśli nie zrobiłeś tego w poprzednim ćwiczeniu, napisz trywialny preconditioner `Precond_I(int N, double **A, double *r, double *p)` przepisujący tablicę reszt wskazywaną przez `r` na tablicę poprawek wskazywaną przez `p`.

A teraz na poważnie

Na tym etapie nigdzie w kodzie nie potrzebujemy macierzy S. Możemy ją całkowicie wyeliminować. Funkcję `Solve` będziemy chcieli jednak używać dla różnych macierzy — dlatego jako argument, zamiast macierzy `double ** A` będziemy przekazywać funkcję mnożenia `void (*mult)(double *, double *)`. Tzn: nagłówek funkcji `Solve` będzie następujący:

```
void Solve(int n, void (*mult)(double *, double *), double *b, double *x0, double *x)
```

A w miejscu mnożenia przez macierz $r = Ax$ będziemy mieli `mult(x,r);`. Teraz funkcję `Solve` będziemy wywoływać przekazując jej konkretną funkcję mnożącą: `Solve(n, SMult, F, d);`.

Na koniec możesz spróbować przerobić funkcję `Solve` tak, aby także preconditioner przyjmowany był jako argument (wskaźnik do funkcji) i wywołać ją z preconditionerem Jacobiego.

Równoległość*

W tej części laboratorium zajmiemy się paralelizacją (zrównolegleniem) napisanego dotychczas kodu. Plik nagłówkowy z funkcjami ułatwiającymi pisanie równoległego kodu:

Plik nagłówkowy ParLib.hpp

Chcielibyśmy teraz wykorzystać fakt, że współczesne procesory posiadają wiele rdzeni. Są one w związku z tym zdolne do wykonywania więcej niż jednego ciągu instrukcji jednocześnie. Nawet bezwiednie korzystamy z tego na co dzień, np. jednocześnie słuchając muzyki i pisząc maile. System operacyjny przydziela wtedy zasoby obliczeniowe (dostęp do rdzeni) do różnych procesów w miarę potrzeby. Rozróżnijmy teraz 2 kluczowe pojęcia:

- Proces: wykonywany program. Blok kontrolny procesu zawiera informację nt. m.in. jego priorytetu, identyfikatora i innych własności. Procesy mają niezależne stosy pamięci i nie komunikują się wzajemnie w wydajny sposób.
- Wątek: część wykonywanego procesu (jeden proces składa się z co najmniej jednego wątku). Wątki danego procesu mają dostęp do jego stosu pamięci i komunikują się wzajemnie w wydajny sposób.

Krótko mówiąc, wątki są „lekkimi" podjednostkami procesu. Kluczowy jest tutaj fakt, że różne wątki jednego procesu mogą jednocześnie wykonywać się na osobnych rdzeniach. Zajmiemy się teraz napisaniem programu, który tworzy kilka wątków i wykonuje obliczenia równoległe.

Zacznijmy od prostego przykładu. Mamy daną stuelementową tablicę `tab`, której wszystkie elementy chcielibyśmy zwiększyć o 1. W tym celu stworzymy 4 wątki, z których każdy zajmie się wydzielonym kawałkiem `tab`. Program wygląda następująco:

```
#include <cstdlib>
#include "ParLib.hpp"

void inc1(double* tab, unsigned int tab_size)
{
    // Numer bieżącego wątku
    const unsigned int id          = self_id();

    // Liczba wszystkich wątków
    const unsigned int n_thr       = no_threads();

    // Bieżący wątek inkrementuje tylko elementy tab o indeksach podziel
    for (int i = id; i < tab_size; i += n_thr)
        tab[i] += 1.;

    return;
}

int main()
{
    // Stwórz tab
    double* tab = (double*)malloc(100 * sizeof(double));

    // Wypełnij tab
    for (int i = 0; i < 100; i++)
        tab[i] = rand();

    // Wywołaj inc1 równolegle
    execute_in_parallel(4, inc1, tab, 100);

    // Zwolnij pamięć
    free(tab);
}
```

Zwróć uwagę, że o liczbie wątków, które zostaną stworzone decyduje dopiero argument funkcji `execute_in_parallel` - funkcja `inc1` napisana jest w sposób ogólny.

Powyższy przykład nie ilustruje jednak głównego problemu programowania równoległego. Wątki działają zupełnie niezależnie - nie muszą się komunikować, a adresy, do których wpisują wartości nie pokrywają się. Rozważmy teraz następujący przykład: mamy funkcję `double skomplikowane_obliczenia(double input)`, która przyjmuje liczbę `input`, wykonuje na niej pracochłonne działania, a następnie zwraca wynik. Chcielibyśmy teraz wywołać ją dla argumentów 3.14 oraz 42.42, a następnie zsumować wyniki. Wywołania dla osobnych argumentów są od siebie niezależne, więc od razu nasuwa się prosty schemat paralelizacji. Naiwna implementacja wygląda następująco:

```
#include <cstdlib>
#include "ParLib.hpp"

double skomplikowane_obliczenia(double input)
{
    // Wyobraźmy sobie, że tutaj dzieją się złożone obliczenia
    return input + 1;
}

// Funkcja wywołująca skomplikowane_obliczenia i dodająca wynik do wkazanego
void fun(double inputs[], double* adres)
{
    // Dany wątek bierze z tablicy inputs argument odpowiadający jego identyf
    const double argument = inputs[self_id()];

    *adres += skomplikowane_obliczenia(argument);  // BŁĄD: race condition
}

int main()
{
    // Wartości wsadowe
    double inputs[2];
    inputs[0] = 3.14;
    inputs[1] = 42.42;

    // Zmienna do której wpisujemy sumę wyników
    double wynik       = 0.;
    double* wynik_ptr  = &wynik;
```

```
    // Wywołaj obliczenia równoległe
    execute_in_parallel(2, fun, inputs, wynik_ptr);
}
```

Problem polega na tym, że oba stworzone wątki będą próbować pisać do tego samego adresu `wynik_ptr`. Tempo wykonywania przez wątki instrukcji nie jest deterministyczne. Może zdarzyć się, że system operacyjny uzna, że inny proces jest ważniejszy od naszego programu i na jakiś czas pozbawi jednego (lub obu) z naszych wątków zasobów. Nie jesteśmy wobec tego w stanie przewidzieć, który z nich skończy pracę pierwszy. W szczególności może zdarzyć się, że oba wątki będą *jednocześnie* starały się zmodyfikować zawartość adresu `wynik_ptr`. W takim wypadku wartość, która będzie finalnie znajdować się pod adresem `wynik_ptr` nie będzie spodziewaną sumą. Taką sytuację nazywamy race condition.

Najpopularniejszym mechanizmem wykorzystywanym do eliminowania race condition są tzw. mutexy (od ang. mutual exclusion). Idea mutexu jest prosta: przed wykonaniem potencjalnie problematycznej operacji, wątek próbuje zablokować mutex (w tym celu komunikuje się z pozostałymi wątkami). Jeżeli mutex jest wolny (tzn. nie został wcześniej zablokowany przez inny wątek), operacja zostaje wykonana, a następnie mutex jest zwalniany. Jeżeli mutex jest zablokowany (przez inny wątek), wtedy wykonanie instrukcji zostaje wstrzymane do momentu zwolnienia mutexu. Taka konstrukcja programu gwarantuje, że co najwyżej jeden wątek będzie wykonywał zestaw potencjalnie problematycznych operacji jednocześnie. Przyjrzyjmy się teraz, jak uniknąć race condition w przypadku powyżej. Modyfikujemy funkcję `fun`:

```
void fun(double inputs[], double* adres)
{
    // Dany wątek bierze z tablicy inputs argument odpowiadający jego id
    const double argument = inputs[self_id()];

    // Blokujemy mutex o numerze 0
    mutex_lock(0);

    *adres += skomplikowane_obliczenia(argument);

    // Zwalniamy mutex po wykonaniu dodawania
    mutex_unlock(0);
}
```


Dodatkowo musimy w funkcji `main` zadeklarować, ile mutexów chcemy używać, także przed wykonaniem `execute_in_parallel` musimy zawołać

```
// Korzystamy tylko z jednego mutexu (o indeksie 0, patrz wyżej)
init_mutex(1);
```

Biblioteka ParLib wspiera używanie wielu mutexów, ich liczbę należy tylko wcześniej zadeklarować przy użyciu `init_mutex`. Korzystając z mutexów należy zachować ostrożnośc i konstruować algorytmy tak, aby wątki się nie zakleszczyły (ang. deadlock), tzn. np. żeby wątek 0 nie czekał na wątek 1, a wątek 1 na wątek 0. W takiej sytuacji wykonanie programu nigdy się nie zakończy.

Zrozumienie powyższych przykładów umożliwi nam przystąpienie do paralelizacji naszej implementacji metody gradientów sprzężonych. Zanim przystąpimy do właściwej pracy, zmierzmy czas, który zajmuje nam rozwiązanie układu $Ax = b$ w sposób szeregowy, aby mieć się do czego później porównać.

Zadanie

Użyj funkcji `tic` i `toc`, aby zmierzyć czas, który zajmuje Twojemu programowi wykonanie funkcji `Solve`. Dobierz `mx` i `my` tak, aby czas ten był „wyczuwalny”, np. kilka sekund. Dzięki temu będziemy mogli miarodajnie kwantyfikować przyspieszenie (lub spowolnienie) kodu wynikające z wielowątkowości.

Zadanie

Zauważmy, że najbardziej kosztowne obliczeniowo jest wykonanie operacji mnożenia macierz-wektor (funkcja `SMult`). Wobec tego, zajmiemy się jej paralelizacją w pierwszej kolejności. Napisz funkcję `Smult_par` (będącą później argumentem `Solve`), która przy użyciu `execute_in_parallel` woła wielowątkowo funkcję `Smult_worker`. Clou tego zadania polega na poprawnej konstrukcji funkcji `Smult_worker`. `Smult_worker` powinna:

- Na podstawie numeru wątku (`self_id`) oraz sumarycznej liczby wątków (`no_threads`) identyfikować zakres elementów, na którym będzie dokonywać mnożenia. Zakresy między wątkami powinny być możliwie równe. Zagadnienie podziału pracy na wiele wątków/procesów nazywa się load balancing.

- Korzystać z mutexów, aby uniknąć jednoczesnego dodawania do wektora wyniku z różnych wątków. Sprawdź, co jest bardziej optymalne: blokowanie całego wektora wyników jednym mutexem, czy blokowanie go kawałkami, tak, aby umożliwić jednoczesny dostęp wątkom piszącym do różnych elementów.

Zadanie

Przyjrzyj się pozostałym elementom funkcji `Solve`. Które z nich można korzystnie zrównoleglić, a które nie? Przetestuj swoje hipotezy posługując się funkcjami `tic` i `toc`.

Dokumentacja biblioteki ParLib

Podstawowa wielowątkowość:

- `void execute_in_parallel(int n_threads, fun, arg1, arg2, ...)` — funkcja generuje `n_threads` wątków. Każny z nich zaczyna wykonywać funkcję `fun`, wywołaną z argumentami `arg1`, `arg2`, itd. Liczba argumentów jest dowolna. Wyjście z funkcji następuje po tym, gdy ostatni ze stworzonych wątków zakończy wykonywanie `fun`. `execute_in_parallel` nie wspiera rekurencji (nie można wołać jej wewnątrz `fun`). Jest to ograniczenie biblioteki ParLib, w „prawdziwych" bibliotekach do programowania równoległego wątki mogą mieć dzieci, wnuki, itd.
- `int self_id()` — funkcja zwracająca numer wątka stworzonego przez `execute_in_parallel`. Każdy wątek stworzony w ten sposób posiada unikatowy numer z zakresu 0 - `n_threads`-1.
- `int no_threads()` — funkcja zwracająca liczbę wątków stworzonych przez `execute_in_parallel`.
- `void init_mutex(int n_mutex)` — funkcja inicjalizująca `n_mutex` mutexów. Można z nich korzystać do końca trwania programu, są później dealokowane automatycznie. `init_mutex` można zawołać jedynie raz.
- `void mutex_lock(int mutex_no)` — funkcja blokująca mutex o numerze `mutex_no`. `mutex_no` musi należeć do zakresu 0 - `n_mutex`-1. Blokowanie mutexu, który został już wcześniej zablokowany przez dany wątek powoduje błąd.
- `void mutex_unlock(int mutex_no)` — funkcja zwalniająca mutex o numerze `mutex_no`. `mutex_no` musi należeć do zakresu 0 - `n_mutex`-1. Zwalnianie mutexu, który został już wcześniej zwolniony przez dany wątek powoduje błąd.

Synchronizacja:

- `void sync()` — funkcja synchronizująca wszystkie wątki stworzone przez `execute_in_parallel`, tzn. każdy wątek zatrzymuje się dopóty, dopóki wszystkie pozostałe wątki nie wywołają `sync()`. Łatwo zauważyć, że wszystkie wątki muszą wywołać `sync()` dokładnie tyle samo razy, w innym wypadku program nigdy nie przestanie się wykonywać.

Pomiar czasu:

- `void tic(unsigned int tic_id)` — przypisuje „stempel czasowy" do identyfikatora `tic_id`. `tic_id` może być dowolne, w przypadku wykonywania wielu pomiarów nie ma obowiązku sekwencyjnego numerowania.
- `double toc(unsigned int toc_id)` — zwraca czas (w sekundach), który upłynął od wywołania `tic(toc_id)`. Jeżeli `tic(toc_id)` nie było wcześniej wołane, zostanie zwrócone 0.

Uwaga: `tic` i `toc` mają domyślny argument 0, także zawołanie

```
tic()
//
// Tutaj skomplikowane obliczenia
//
toc()
```

zwróci czas wykonania obliczeń.