Programowanie równoległe. Przetwarzanie równoległe i rozproszone.

Laboratorium 10

Cel:

- pogłębienie umiejętności pisania programów równoległych w środowisku OpenMP

Zajęcia:

1. Utworzenie katalogu roboczego (np. *lab_10*) i podkatalogu (*zmienne*).
2. Skopiowanie paczki *openmp_watki_zmienne.tgz*, rozpakowanie w katalogu roboczym, uruchomienie programu
   a) opcja kompilacji *-g* ma na celu uniemożliwienie optymalizacji, które mogłyby zmienić sposób działania kodu w stosunku do zapisu z kodu źródłowego
3. Poprawienie czytelności wydruku przez zastosowanie odpowiednich dyrektyw OpenMP (każdy wątek drukuje bez ingerencji innych); wydruki zorganizować tak, żeby wydruki zmiennych wspólnych w każdym wątku były takie same jak wydruki po wyjściu z obszaru równoległego
4. Uruchomienie programu z domyślną liczbą wątków – sprawdzenie poprawności nadawania wartości zmiennym
5. Sprawdzenie (nie)poprawności działania programu w wersji wielowątkowej – przy poprawnym działaniu wartości zmiennych wspólnych (z klauzuli *shared)* po wyjściu z obszaru równoległego powinny być identyczne przy każdym uruchomieniu (zgodnie z wymaganiem determinizmu działania)
   a) w wersji dostarczonej program nie jest deterministyczny,
   b) aby to wykazać należy uruchamiać go z rosnącą liczba wątków (nadawaną za pomocą wartości zmiennej środowiskowej, bez rekompilacji kodu!), aż do takiej liczby (10?, 100? - to zależy od maszyny i środowiska wykonania) przy której pojawią się błędy – różne wartości zmiennych wspólnych na zakończenie dla różnych wykonań programu
6. Poprawienie błędów w działaniu kodu:
   a) dla zmiennej określonej jako *a_shared* ochrona w sekcji krytycznej może obejmować większy fragment kodu (np. całą pętlę)
   b) dla zmiennej określonej jako *e_atomic* intencją jest zapewnienie poprawności działania przez użycie operacji atomowych
   c) w kodzie znajduje się także mniej oczywista zależność efektu działania od kolejności wykonywania operacji przez wątki – WAR dla zmiennej *a_shared*, co może prowadzić do różnych wartości zmiennej *d_local_private* w różnych uruchomieniach programu i różnych wątkach (założeniem jest, że wartość ta powinna być taka sama w każdym wątku i przy każdym uruchomieniu programu) – usunięcie zależności wymaga wprowadzenia bariery
7. Uruchomienie po poprawkach i pokazanie poprawnego działania dla liczby wątków prowadzącej uprzednio do błędów wykonania **(ocena)**
8. Napisanie drugiego obszaru równoległego, bezpośrednio po pierwszym. Przetestowanie funkcjonowania dyrektywy *threadprivate* – zmienne objęte tą dyrektywą mają zachować swoje prywatne wartości w kolejnych obszarach równoległych
   a) zdefiniowanie przykładowej zmiennej *f_threadprivate* i objęcie jej dyrektywą *threadprivate*
   b) nadanie zmiennej *f_threadprivate* w pierwszym obszarze równoległym wartości identyfikatora wątku
   c) wydrukowanie przez wszystkie wątki wartości zmiennej *f_threadprivate* w drugim obszarze równoległym
9. Uruchomienie programu za pomocą 5 wątków korzystając z funkcji bibliotecznej ustalającej ich liczbę – w przypadku użycia dyrektywy *threadprivate* wszystkie obszary mają mieć taką samą liczbę wątków – wywołanie funkcji powinno znaleźć się przed pierwszym obszarem równoległym **(ocena)**
10. Utworzenie podkatalogu roboczego (np. *lab_10/pde*).
11. Skopiowanie pliku *openmp_zaleznosci.c*, rozpakowanie w katalogu roboczym, uruchomienie programu

a) kompilacja musi uwzględniać opcję *-fopenmp* (dla *gcc*) i dołączenie biblioteki matematycznej *-lm*

12. Analiza zależności przenoszonych w pętli (opis powinien znaleźć się w sprawozdaniu)
13. Poprawne zrównoleglenie pętli, przez wprowadzenie dodatkowej tablicy, której użycie pozwala usunąć zależności
14. Uruchomienie programu za pomocą 2 wątków korzystając z odpowiedniej klauzuli dyrektywy *parallel*, sprawdzenie poprawności działania kodu. **(ocena)**

---------- 3.0 ----------

15. Utworzenie podkatalogu roboczego (np. *lab_10/search_max*)
16. W podkatalogu rozpakowanie i uruchomienie programu wyszukiwania wartości maksymalnej w tablicy
    a) obserwacja sposobu obliczania maksimum dla funkcji z pliku *search_max_openmp.c:*
    - w klasycznej wersji sekwencyjnej – *search_max*
    - w standardowej wersji zrównoleglenia pętli *for* - *search_max_openmp_simple*
17. Uzupełnienie funkcji *search_max_openmp_task* obliczania maksimum w wersji równoległej OpenMP z wykorzystaniem zadań
    a) obserwacja mechanizmu tworzenia zadań w obszarze równoległym, przez pojedynczy wątek (dyrektywa *single*)
    b) dodanie definicji pojedynczego zadania (*task*)
    - w dyrektywie *task* należy użyć klauzuli *default(none)* i ustalić jak poprawnie i optymalnie przeprowadzać obliczenia (jakich zmiennych użyć)
    - do definicji zadania można wykorzystać funkcję sekwencyjną *search_max* lub cały kod obliczeń umieścić w definicji zadania
18. Uruchomienie programu, sprawdzenie poprawności działania i porównanie czasów wykonania poszczególnych wersji **(ocena)**
19. Utworzenie podkatalogu roboczego (np. *lab_10/sortowanie*)
20. W podkatalogu roboczym rozpakowanie i uruchomienie dostarczonego programu realizującego sortowanie sekwencyjne i równoległe
    a) obserwacja czasów wykonania dla różnych algorytmów sortowania w implementacji sekwencyjnej i równoległej
    b) najlepiej dobrać rozmiar tablicy, tak aby czas sortowania przekraczał 0.1 s
    - dzięki temu w wersjach równoległych narzut systemowy może stać się pomijalny
21. Analiza technik zrównoleglenia sortowania przez scalanie za pomocą dyrektyw OpenMP w funkcjach w pliku *merge_sort_openmp.c* – krótki opis technik powinien znajdować się w sprawozdaniu
    a) w funkcji *merge_sort_openmp* zastosowana jest równoległość z tworzeniem zadań (dyrektywa *task*)
    - *merge_sort_openmp_2* stosuje te same techniki, dodatkowo wypisując dla kilku pierwszych poziomów zagnieżdżenia wywołań rekurencyjnych numer poziomu i identyfikator wykonującego wątku
    b) w funkcji *merge_sort_openmp_4* zastosowana jest równoległość ze zrównolegleniem wykonania fragmentów kodu (dyrektywa *sections*)
    - konieczne jest ograniczenie tworzenia nowych wątków (poprzez zagnieżdżone dyrektywy *parallel*) do pewnego momentu (przekazywana zmienna poziom, określająca kolejne kroki podziału tablicy)
    c) należy opisać użyte dyrektywy i ich klauzule – każdorazowo należy określić znaczenie dyrektywy i klauzuli
22. Zadaniem jest modyfikacja funkcji *merge_sort_openmp_2*, tak aby zwiększyć jej wydajność w stosunku do pierwotnej wersji, będącej uproszczoną wersją funkcji *merge_sort_openmp*
    a) należy wprowadzić klauzulę final w celu ograniczenia liczby tworzonych zadań
    b) dodatkowo należy wywołać optymalną wersję sekwencyjną (np. *sortowanie_szybkie*) w momencie, kiedy nie są już tworzone nowe zadania

---------- 4.0 ----------

23. Powrót do katalogu *search_max*. Uzupełnienie programu o definicje zadań (*tasks*) – dla wersji równoległej OpenMP wyszukiwania binarnego (można wzorować się na procedurze sortowania przez scalanie *merge_sort_openmp* z programu sortowania) – należy założyć tworzenie pełnego drzewa wywołań binarnych, aż do momentu, kiedy podtablica ma już tylko jeden element (także w tym przypadku w dyrektywie *task* użyć klauzuli *default(none)*) **(ocena)**
24. Zmodyfikowanie programu wyszukiwania binarnego, tak żeby po przekroczeniu pewnego poziomu w drzewie wywołań nie były generowane dalsze zadania, a algorytm korzystał z szybkiego wyszukiwania liniowego (można wzorować się na procedurze sortowania przez scalanie *merge_sort_openmp_2* z programu sortowania) **(ocena)**

---------- 5.0 ----------

25. Utworzenie podkatalogu roboczego (np. *lab_10/mat_vec*) .
26. Skopiowanie paczki *openmp_mat_vec.tgz*, rozpakowanie w katalogu roboczym, uruchomienie programu.
27. Napisanie 4 wersji zrównoleglenia algorytmu mnożenia macierz-wektor (w każdym z przypadków należy zrównoleglić procedury zawarte w pliku *mat_vec.c*, a wynik ma być identyczny, jak obliczany w pliku *openmp_mat_vec.c* w pętlach testowych)
    a) każdorazowo stosowana jest jako dana wejściowa ta sama tablica *a*, raz traktowana jako macierz zapisana wierszami (*row major* - kolejne wyrazy w wierszu są kolejnymi elementami tablicy), raz jako macierz zapisana kolumnami (*column major* - kolejne elementy w kolumnie stanowią kolejne wyrazy w tablicy)
        - w efekcie mimo że tablica jest w programie taka sama, to odpowiada dwóm różnym macierzom (są one swoimi transpozycjami - a jako że nie są symetryczne, są więc różne)
    b) 4 wersje równoległego algorytmu odpowiadają 2 formom zapisu (przechowywania) i 2 sposobom dekompozycji:
        - *mat_vec_row_row_decomp* – dekompozycja wierszowa dla macierzy przechowywanych wierszami (*row major*) **(ocena)**
        - *mat_vec_row_col_decomp* – dekompozycja kolumnowa dla macierzy przechowywanych wierszami (*row major*), ze zrównolegleniem pętli po kolumnach jako pętli wewnętrznej i klauzulą *reduction* **(ocena)**
        - *mat_vec_col_col_decomp* – dekompozycja kolumnowa dla macierzy przechowywanych kolumnami *(column major)*, ze zrównolegleniem pętli po kolumnach jako pętli zewnętrznej i z prywatnymi wektorami *y_local* dla każdego wątku, sumowanymi do ostatecznego wektora *y* w sekcji krytycznej poza pętlą zewnętrzną – wymaga to użycia osobno dyrektyw *parallel i for* **(ocena)**
        - *mat_vec_col_row_decomp* – dekompozycja wierszowa dla macierzy przechowywanych kolumnami *(column major)*, ze zrównolegleniem pętli po wierszach jako pętli wewnętrznej (nie wymaga synchronizacji, redukcji itp.) **(ocena)**

Warunki zaliczenia:

1. Obecność na zajęciach i wykonanie co najmniej kroków 1-14
2. Oddanie sprawozdania o treści i formie zgodnej z regulaminem ćwiczeń laboratoryjnych – z krótkim opisem zadania (cel, zrealizowane kroki), kodem źródłowym procedur w C, obrazami wydruków z wykonania programów i wnioskami – **wnioski powinny zawierać krótkie omówienie zastosowanych dyrektyw i klauzul oraz ich roli w kodzie**