Logowanie do serwisu | Użytkownik: AA APlaceHolder | Kod: | Login...

Programowanie Rozproszone i Równoleg
Studia niestacjonarne

Strona z zadaniami + interfejs systemu wysyłania rozwiązań i pytań

Możliwość zadawania pytań zostanie włączona około 19 listopada

Uwagi dotyczące wysyłania rozwiązań dostępne są przez ten link.

Proszę się z nimi zapoznać!

Zadanie 01 termin V. Do zdobycia maksymalnie: 0.5pkt.

Zadanie - poszukiwanie

Idea

Trzeba odszukać maksima lokalne pewnej funkcji. Problem w tym, że samo zagadnienie odszukania optimum funkcji jest już problemem, a my chcemy odkryć więcej niż jedno maksimum i to w pojedynczym uruchomieniu programu.

Program działa symulując zachowanie roju maszyn (robotów), które mogą się ze sobą komunikować. Aby rój był w stanie odkryć wiele rozwiązań, musi podzielić się na części. Osiąga się to poprzez ograniczenie zasięgu wymiany informacji. Każdy z robotów posiada antenę, której zasięg można zmieniać tak, aby w sąsiedztwie robota znajdowała się ograniczona liczba innych robotów.

Działanie programu w praktyce

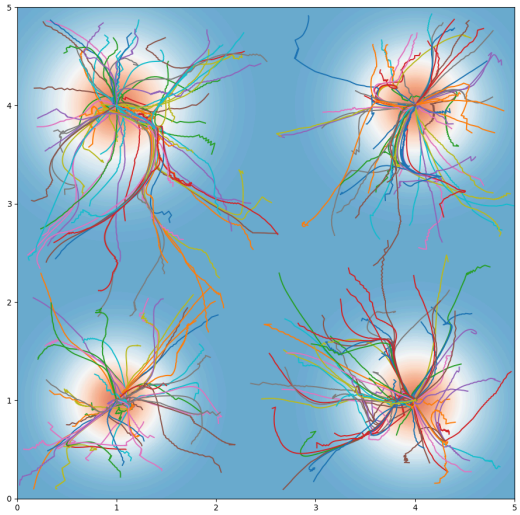
Poniżej znajdują się ilustracje działania programu. Program testowany był za pomocą roju o rozmiarze 1500 sztuk robotów. Funkcją testową była suma funkcji typu:

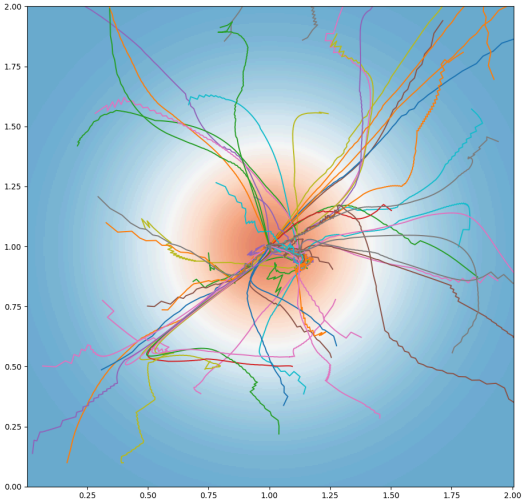
```
A exp( -r*r / B )
```

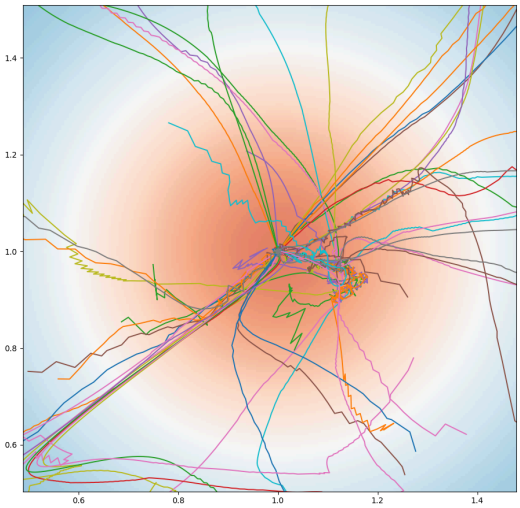
Gdzie:

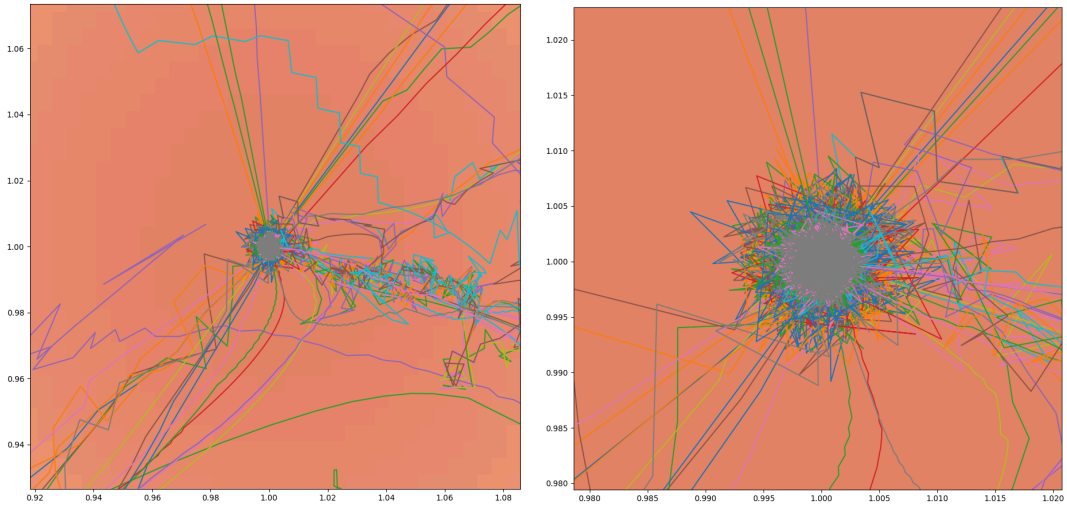
- A - maksymalna wartość funkcji
- r - odległość od maksimum do miejsca, w którym badamy wartość funkcji
- B - parametr określający szybkość obniżania się wartości funkcji wraz z odległością

Dokładne współczynniki można odszukać w pliku `main.cpp`.









0:00 / 0:12

Na powyższych obrazkach pokazana została droga pokonana w trakcie pracy programu przez pewną liczbę robotów (na rysunku mamy zaznaczone trasy tylko części ze wszystkich biorących udział w optymalizacji). Animacja pokazuje jak roboty dzielą się na grupy i podążają w kierunku lokalnych maksimów badanej funkcji. Uwaga: roboty poruszały się w 3D, obrazki pokazują rzut na płaszczyznę - stąd "dziwne" zachowanie niektórych robotów.

Wersja sekwencyjna kodu

Otrzymujecie Państwo gotowe, działające rozwiązanie problemu optymalizacji. Klasa `SequentialSwarm` zawiera kod sekwencyjny. Dziedziczy ona po `Swarm`. Wszystkie metody `Swarm` są wirtualne i nie mają na tym poziomie implementacji.

Algorytm działania programu jest bardzo prosty i polega na cyklicznym wykonaniu kilku kroków:

1. Dla każdego robota wyznaczana jest wartość funkcji, której maksimum jest poszukiwane.
2. Każdy robot sprawdza ile robotów znajduje się w zasięgu jego anteny i wśród nich wyszukuje takiego, którego wartość funkcji jest największa.
3. Robot przesuwa się w kierunku sąsiedniego robota o największej wartości funkcji. Jeśli lepsza wartość nie została znaleziona, robot nie zmienia położenia.
4. Na podstawie liczby wykrytych sąsiadów ustalany jest zasięg anteny w kolejnej iteracji.

Zadanie

Zadanie polega na napisaniu kodu, który używając MPI przyspieszy obliczenia.

Wymagania dla aplikacji równoległej

- Wersja równoległa ma bazować na dostarczonym rozwiązaniu sekwencyjnym.
- Wersja równoległa i sekwencyjna muszą dawać ten sam wynik.
- Dodatkowe procesy mają przyspieszać obliczenia.
- Wszystkie procesy mają uczestniczyć w obliczeniach. Czyli, już dwa procesy mają przyspieszyć rachunek.
- Program ma działać efektywnie - tj. dostając do dyspozycji kolejne rdzenie/procesory ma skracać czas potrzebny do uzyskania rezultatu. Platformą referencyjną do testów efektywności jest wyłącznie klaster.
- Kod musi dać się skompilować i uruchomić na udostępnianych komputerach Wydziałowych (klaster). Do własnych testów proszę używać MPI w wersji OpenMPI.
- Zadanie polega na poprawie efektywności działania programu za pomocą MPI, nie zaś optymalizacji kodu. Sprowadza się to do żądania aby czas na jaki program zajmuje CPU był zbliżony dla obu wersji kodu. Czyli, jeśli wersja sekwencyjna potrzebuje X sekund CPU na wykonanie zadania, to wersja równoległa powinna zrealizować to samo zadanie używając tyle samo czasu CPU, choć w przypadku kodu MPI czas rzeczywisty powinien być odpowiednio krótszy (dla 100% efektywności czas rzeczywisty powinien wynosić X/P, gdzie P to liczba procesów).
  Wyjaśnienie aby nie było wątpliwości: Czas rzeczywisty i CPU mogą się od siebie różnić, bo program MPI używa zasobów wielu procesorów i ich rdzeni równocześnie. Jeśli mam procesor z 4-oma rdzeniami, to w ciągu 60 sekund czasu rzeczywistego program działający na tym procesorze może zrealizować obliczeniach zajmujące 240 sekund (4 rdzenie zajęte przez 60 sekund każdy).
- Państwa program będzie musiał zmierzyć się z kodem oryginalnym, ale też z sobą samym pracując z różnymi ustawieniami oraz przy różnej liczbie dostępnych procesorów/węzłów klastra. Podstawowa kwestia to odpowiedz na pytanie: czy program efektywnie używa otrzymanych zasobów sprzętowych?

Uwagi

- Szybkość komunikacji pomiędzy procesami w obrębie jednego komputera jest wielokrotnie wyższa od tej, którą można uzyskać używając klastra. Aspekt komunikacji należy przemyśleć. Zdecydowanie trzeba ustalić co, kiedy i komu trzeba przekazywać.
- Jeden większy komunikat będzie typowo szybszy od wielu mniejszych, które przekażą te same dane.
- Myśląc o przyspieszeniu pracy programu należy przyjąć, że w algorytmie pierwsze dwa kroki będą zajmować najwięcej czasu. Funkcja wyliczana na podstawie położenia każdego z robotów może być skomplikowana. Wyszukiwanie sąsiadów wymaga N*(N-1) operacji, gdzie N to liczba robotów.
- Idea rozwiązania to podział pracy pomiędzy procesy. Każdy proces musi obsługiwać cześć robotów. W ten sposób można skrócić obliczenia. Ale uwaga: po wykonaniu każdego kroku położenia robotów się zmieniają. Aby możliwe było wykonanie kolejnego kroku w wielu procesach, muszą one najpierw wymienić się potrzebnymi informacjami.
- Proszę nie zmieniać pozostałych fragmentów kodu (innych klas). Państwa rozwiązanie dotyczy wyłącznie jednej klasy (`ParallelSwarm`). Stosowane implementacje badanej funkcji, kodu `main.cpp`, stałych w `consts.h`, zachowania anteny itd. mogą ulec zmianie. Program ma działać poprawnie w każdych warunkach. Liczba wymiarów, w których poruszają się roboty także może ulec zmianie. Staramy się napisać narzędzie, a nie program, który potrafi poprawnie działać tylko w określonym, jednym przypadku.
- `Main.cpp` pokazuje ideę użycia procesów (kolejność wywoływania metod Swarm). Proszę zauważyć, że nie każda metoda wykonywana jest dla wszystkich procesów. W szczególności, położenia startowe zna tylko proces 0.

Kompilacja kodu

W pliku `main.cpp` jest definicja MPI_ON. Po ustawieniu:

```
#define MPI_ON
```

Kod będzie działać z użyciem MPI.

Kod w wersji MPI najprościej skompilować tak:

```
mpiCC -O *cpp
```

Dostarczanie rozwiązania:

- Rozwiązanie (klasa `ParallelSwarm`) ma rozszerzać (dziedziczyć) klasę `Swarm`. Do własnych testów w kodzie `Main.cpp` należy wymienić udostępnioną implementację z `SequentialSwarm` na `ParallelSwarm`. Idea użycia `Swarm` zostanie w trakcie testów zachowana.
- Klasa `ParallelSwarm` ma posiadać następujący konstruktor:

```
ParallelSwarm( int robots, Antenna *antenna, Function *function  );
```

- Nagłówek o nazwie `ParallelSwarm.h` proszę również dostarczyć.
- Pliki .cpp i .h proszę wgrywać do systemu osobno.
- Uwaga: jeśli ktoś z Państwa chce, w ramach jednego terminu, dostarczyć więcej niż jedno rozwiązanie, to proszę zadbać o ponowne wgranie zarówno plików .cpp, jak i odpowiedniego pliku .h - nazwy tych plików muszą do siebie pasować. Czyli np. SequentialSwarm2.cpp i SequentialSwarm2.h Proszę jednak w "include" nadal używać SequentialSwarm.h (pliki otrzymają odpowiednie nazwy przed ich kompilacją).
- Warto zadbać o to, aby Państwa kod nie wyświetlał komunikatów na terminalu. Coś takiego może znacząco pogorszyć efektywności pracy programu.
- Proszę za pomocą "include" wskazywać wyłącznie pliki nagłówkowe.

Linki:

- Link do kodu źródłowego: - tutaj -

Test

Udostępniam wersję Main.cpp i consts.h, które używane były w testach.

Testy prowadzone były dla następującego układu komputerów/procesów:

```
komputery    procesy
 1           2
 1           5
 3           6
 3           9
 3           12
 4           4
 4           8
```

Najpierw wykonywane były testy dla jednego komputera. Jeśli te zostały zaliczone kod uruchamiany był na klastrze.

Ocena zależy od średniej ważonej z efektywności obliczeń na klastrze. Aby była ona bardziej wiarygodna czas obliczeń na klastrze został wydłużony poprzez zwiększenie liczby kroków symulacji do 500 (na klastrze używanym do testów są inne ograniczenia na maksymalny czas pracy zadań) - inne parametry pozostały bez zmian.

W trakcie wyznaczania efektywności pojawił się problem programów, w których oryginalny kod został na tyle zmodyfikowany, że programy uzyskiwały efektywność powyżej 100%. W takich przypadkach efektywność była skalowana względem skróconego czasu wykonania programu sekwencyjnego. Chodziło o to, aby ocena była wynikiem przyspieszenia pracy programu za pomocą MPI, nie zaś optymalizacji kodu.

Punktacja

```
                      I termin  II termin  III termin  IV termin
Efektywność od 40%    0.5pkt    0.5pkt     0.5pkt      0.5pkt
Efektywność od 50%    2.0pkt    1.0pkt     0.7pkt      0.5pkt
Efektywność od 60%    2.5pkt    1.5pkt     0.8pkt      0.6pkt
Efektywność od ok. 64% 2.6pkt   1.6pkt     0.8pkt      0.6pkt
```

```
Efektywność od ok. 66% 2.7pkt    1.7pkt     0.8pkt      0.6pkt
Efektywność od ok. 69% 2.8pkt    1.8pkt     0.9pkt      0.7pkt
Efektywność od ok. 71% 2.9pkt    1.9pkt     0.9pkt      0.7pkt
Efektywność od ok. 75% 3.0pkt    2.0pkt     1.0pkt      0.7pkt
```

- Link do kodu źródłowego plików użytych w teście: - tutaj -

Zmieniłem lekko sposób testowania zgodności pozycji. Uwzględnia on pojawiające się czasami pozycje ujemne.

Punktacja: do 0.5pkt. | Zadawanie pytań do: 2025-01-31 | Nadsyłanie rozwiązań do: 2025-02-05

Do zadania zgłoszono pytań: 2, a liczba odpowiedzi to: 2 | LINK do bazy pytań/odpowiedzi.

Możliwość dostarczania rozwiązań włączy się automatycznie po zakończeniu okresu zadawania pytań.

Zadanie 02 termin IV. Do zdobycia maksymalnie: 0.7pkt.

Idea zadania

W zadaniu tym chodzi o współbieżne wykonywanie prostych operacji i przekazywanie danych pomiędzy wątkami.

Co trzeba zrobić?

W tym zadaniu otrzymacie Państwo dwuwymiarową tablice liczb całkowitych. Dostęp do pól (odczyt i zapis) będzie możliwy wyłącznie poprzez metody stosownego interfejsu. W tablicy znajdują się różne liczby nieujemne. Problem polega na tym, aby współbieżne przeszukując tablicę wykryć wszystkie takie pary sąsiednich pozycji w tablicy, których suma zapisanych w tablicy wartości równa jest podanej liczbie. W miejsce takiej pary do tablicy należy zapisać 0. Zbiór wszystkich par pozycji w tablicy jest dodatkowym wynikiem pracy programu. Na pewno pary, które należy usunąć nie będą ze sobą bezpośrednio sąsiadować.

Przykładową tablice wraz z zaznaczonymi znalezionymi parami sąsiednich pozycji, których komórki dają w sumie 10 pokazano na poniższym obrazku. Po pozycjach znalezionych par nie zostały tu wpisane zera. Każda para zaznaczona została innym kolorem tła.

1	1	3	3	2	7	11	5
0	0	1	5	1	3	2	3
0	3	1	1	11	41	0	8
9	0	5	5	0	1	12	5
1	3	4	11	11	1	3	3

Praca współbieżna

Poniżej podana jest lista oczekiwanego zachowania programu pod względem współbieżności.

1. Wątki, których program będzie używał do operacji na tablicy należy utworzyć za pomocą otrzymanej fabryki wątków.
2. Do realizacji odczytów można będzie otrzymać kilka wątków, do realizacji zapisu (wpisywania zer) tylko jeden wątek. Wszystkich tych wątków należy używać.
3. Z tablicy należy odczytywać dane współbieżnie wszystkimi otrzymanymi wątkami.
4. Musi udać się stwierdzić jednoczesne czytanie danych przez więcej niż jeden wątek odczytujący.
5. Tej samej pozycji z tablicy nie wolno czytać wielokrotnie
6. Odczyt/zapis na tej samej pozycji nie mogą zachodzić w tym samym czasie (logika zadania sama z siebie wyklucza już taką sytuację).
7. Odkryte pary pozycji prowadzących do określonej sumy muszą być zastępowane zerami na bieżąco (np. nie wolno czekać na zakończenie procesu odczytywania wszystkich danych). Jeśli para pozycji zostanie odkryta, do pozycji tablicy wchodzących w skład pary mają być możliwie szybko zapisane zera. Przypominam: zera wpisuje jeden wydzielony do tego celu wątek.
8. Program musi wykryć wszystkie pary pozycje dające określoną sumę.
9. Wątki czytające dane z tablicy mogą zacząć pracę dopiero po wykonaniu metody `start`.
10. Program musi przetworzyć wszystkie pozycje tablicy.
11. Program musi wykonywać pracę wszystkimi wątkami. Ilość wykonanej pracy (liczba odczytów) powinna być dla każdego wątku zbliżona
12. Wraz z wątkiem przekazywana będzie pozycja startowa, z której wątek ma rozpocząć proces przeszukiwania tablicy.
13. Wątek ma badać sąsiednie pozycje. Niedopuszczalne jest przeskakiwanie pozycji. Czyli jeśli pewien wątek zbadał już pozycję (x,y), to w kolejnym odczycie może on zbadać pozycje różniące się od (x,y) o 1 - zakładamy, że położenie np. (x+1,y+1) jest także położeniem sąsiednim. Pozycja (x,y) ma 8 pozycji sąsiednich. Inaczej: jeśli aktualny odczyt dotyczy pozycji (x,y), to wcześniej musiał zostać zrealizowany odczyt na jednej z pozycji sąsiednich.
14. Wynik pracy programu musi być możliwy do odebrania bezpośrednio po zakończeniu odczytu wszystkich pozycji i po wpisaniu do nich 0.
15. Program ma pracować z rozsądną wydajnością. Po to otrzymuje więcej niż jeden wątek z prawem odczytu, aby z zasobów tych korzystać.
16. Metoda `start` uruchamiająca przeszukiwanie tablicy ma działać nieblokująco. Wątek, który ją wykona ma wrócić do realizacji swoich zadań. O zakończeniu przeszukiwania tablicy dowie się poprzez okresowe wykonanie metody `result`. Metoda `result` zwraca pusty zbiór jako informacja o trwaniu pracy.
17. Po zwróceniu niepustego wynikowego zbioru program nie ma prawa wykonywać jakichkolwiek operacji na tablicy a wszystkie używane przez program wątki mają być w stanie `TERMINATED`.
18. Wątek modyfikujący stan tablicy w trakcie oczekiwania na dane (pozycje do ustawienia na 0) nie może używać CPU (ma być uśpiony). Zadaniem tego wątku jest wyłącznie wstawianie 0 do tablicy.
19. Ta sama para pozycji **może** zostać umieszczona w zbiorze dwukrotnie - przykładowo dopuszczalne jest pojawienie się par: {(5,5),(4,4)} i {(4,4),(5,5)}.
20. Ta sama pozycja **może** być dwukrotnie zapisywana zerami (patrz poprzedni przykład z dwoma parami różniącymi się kolejnością położeń).

Dodatkowe założenia/uwagi

- Jeśli oczekiwana suma to X, to X nie pojawi się w żadnej z komórek tablicy.
- Rozwiązanie będzie jednoznaczne. Jeśli oczekiwana suma to np. 10 to nie pojawią się w tablicy 3 sąsiednie komórki o wartości 5.
- Pozycje startowe dla poszczególnych wątków będą od siebie odseparowane. Przykładowo: dla zaprezentowanego obrazka dwa wątki mogą uzyskać położenia: (2,3) i (6,2), gdzie pierwsza liczba to kolumna, druga to wiersz.
- Tablica będzie posiadać co najmniej jedną parę pozycji stanowiących rozwiązanie.
- Przed wykonaniem metody `start` wykonane zostaną metody `setThreadsFactory` oraz `setTable`.
- Fabryka wątków dostarczać będzie je w stanie nieuruchomionym. Metodę `start` należy wykonać samodzielnie.

Dostarczanie rozwiązania

Proszę o dostarczenie kodu **źródłowego** klasy `ParallelExplorer`. W klasie można umieścić własne metody i pola. Klasa ma implementować interfejs `Explorer`.

Plik z rozwiązaniem może zawierać inne klasy, ale tylko klasa `ParallelExplorer` może być publiczna.

Kodu, który sam dostarczam nie wolno modyfikować. W trakcie testów używany będzie w takiej formie, w jakiej został udostępniony. Proszę nie dołączać go do rozwiązań. Będzie dostępny w trakcie testów.

Programy będą testowane za pomocą czystego środowiska Java w wersji 21.

Linki

- Dokumentacja
- Kod źródłowy

Punktacja: do 0.7pkt. | Zadawanie pytań do: 2025-01-31 | Nadsyłanie rozwiązań do: 2025-02-02

Do zadania zgłoszono pytań: 8, a liczba odpowiedzi to: 8 | LINK do bazy pytań/odpowiedzi.

Możliwość dostarczania rozwiązań włączy się automatycznie po zakończeniu okresu zadawania pytań.

Zadanie 02 termin V. Do zdobycia maksymalnie: 0.5pkt.

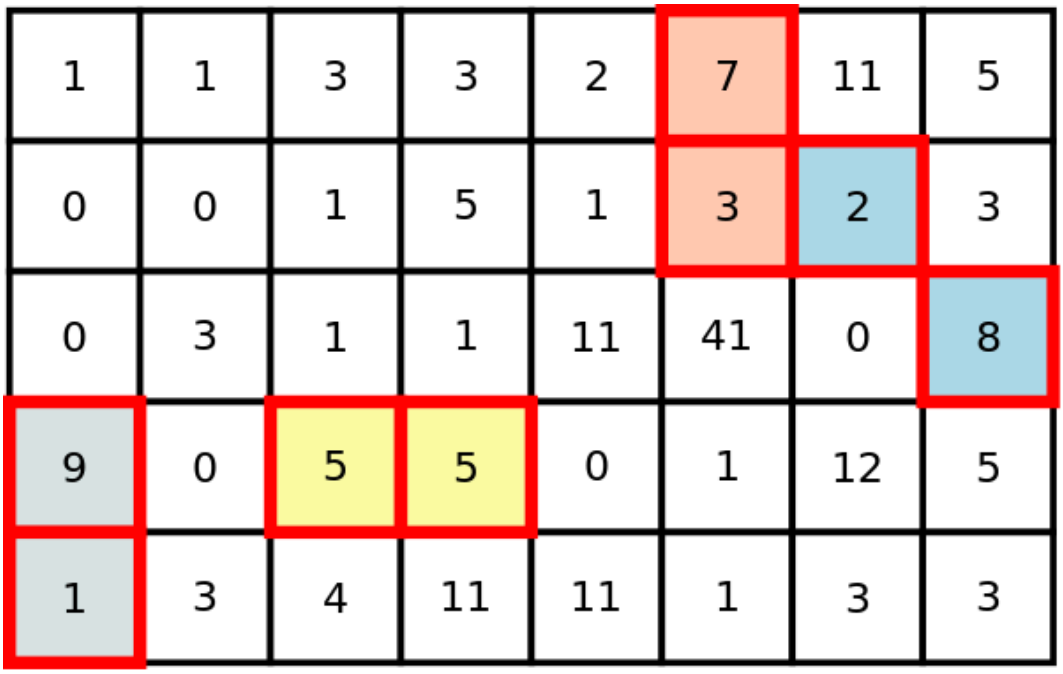
Idea zadania

W zadaniu tym chodzi o współbieżne wykonywanie prostych operacji i przekazywanie danych pomiędzy wątkami.

Co trzeba zrobić?

W tym zadaniu otrzymacie Państwo dwuwymiarową tablice liczb całkowitych. Dostęp do pól (odczyt i zapis) będzie możliwy wyłącznie poprzez metody stosownego interfejsu. W tablicy znajdują się różne liczby nieujemne. Problem polega na tym, aby współbieżne przeszukując tablicę wykryć wszystkie takie pary sąsiednich pozycji w tablicy, których suma zapisanych w tablicy wartości równa jest podanej liczbie. W miejsce takiej pary do tablicy należy zapisać 0. Zbiór wszystkich par pozycji w tablicy jest dodatkowym wynikiem pracy programu. Na pewno pary, które należy usunąć nie będą ze sobą bezpośrednio sąsiadować.

Przykładową tablice wraz z zaznaczonymi znalezionymi parami sąsiednich pozycji, których komórki dają w sumie 10 pokazano na poniższym obrazku. Po pozycjach znalezionych par nie zostały tu wpisane zera. Każda para zaznaczona została innym kolorem tła.



Praca współbieżna

Poniżej podana jest lista oczekiwanego zachowania programu pod względem współbieżności.

1. Wątki, których program będzie używał do operacji na tablicy należy utworzyć za pomocą otrzymanej fabryki wątków.
2. Do realizacji odczytów można będzie otrzymać kilka wątków, do realizacji zapisu (wpisywania zer) tylko jeden wątek. Wszystkich tych wątków należy używać.
3. Z tablicy należy odczytywać dane współbieżnie wszystkimi otrzymanymi wątkami.
4. Musi udać się stwierdzić jednoczesne czytanie danych przez więcej niż jeden wątek odczytujący.
5. Tej samej pozycji z tablicy nie wolno czytać wielokrotnie
6. Odczyt/zapis na tej samej pozycji nie mogą zachodzić w tym samym czasie (logika zadania sama z siebie wyklucza już taką sytuację).
7. Odkryte pary pozycji prowadzących do określonej sumy muszą być zastępowane zerami na bieżąco (np. nie wolno czekać na zakończenie procesu odczytywania wszystkich danych). Jeśli para pozycji zostanie odkryta, do pozycji tablicy wchodzących w skład pary mają być możliwie szybko zapisane zera. Przypominam: zera wpisuje jeden wydzielony do tego celu wątek.
8. Program musi wykryć wszystkie pary pozycje dające określoną sumę.
9. Wątki czytające dane z tablicy mogą zacząć pracę dopiero po wykonaniu metody `start`.
10. Program musi przetworzyć wszystkie pozycje tablicy.
11. Program musi wykonywać pracę wszystkimi wątkami. Ilość wykonanej pracy (liczba odczytów) powinna być dla każdego wątku zbliżona
12. Wraz z wątkiem przekazywana będzie pozycja startowa, z której wątek ma rozpocząć proces przeszukiwania tablicy.
13. Wątek ma badać sąsiednie pozycje. Niedopuszczalne jest przeskakiwanie pozycji. Czyli jeśli pewien wątek zbadał już pozycję (x,y), to w kolejnym odczycie może on zbadać pozycje różniące się od (x,y) o 1 - zakładamy, że położenie np. (x+1,y+1) jest także położeniem sąsiednim. Pozycja (x,y) ma 8 pozycji sąsiednich. Inaczej: jeśli aktualny odczyt dotyczy pozycji (x,y), to wcześniej musiał zostać zrealizowany odczyt na jednej z pozycji sąsiednich.
14. Wynik pracy programu musi być możliwy do odebrania bezpośrednio po zakończeniu odczytu wszystkich pozycji i po wpisaniu do nich 0.
15. Program ma pracować z rozsądną wydajnością. Po to otrzymuje więcej niż jeden wątek z prawem odczytu, aby z zasobów tych korzystać.
16. Metoda `start` uruchamiająca przeszukiwanie tablicy ma działać nieblokująco. Wątek, który ją wykona ma wrócić do realizacji swoich zadań. O zakończeniu przeszukiwania tablicy dowie się poprzez okresowe wykonanie metody `result`. Metoda `result` zwraca pusty zbiór jako informacja o trwaniu pracy.
17. Po zwróceniu niepustego wynikowego zbioru program nie ma prawa wykonywać jakichkolwiek operacji na tablicy a wszystkie używane przez program wątki mają być w stanie `TERMINATED`.
18. Wątek modyfikujący stan tablicy w trakcie oczekiwania na dane (pozycje do ustawienia na 0) nie może używać CPU (ma być uśpiony). Zadaniem tego wątku jest wyłącznie wstawianie 0 do tablicy.

19. Ta sama para pozycji **może** zostać umieszczona w zbiorze dwukrotnie - przykładowo dopuszczalne jest pojawienie się par: {(5,5),(4,4)} i {(4,4),(5,5)}.
20. Ta sama pozycja **może** być dwukrotnie zapisywana zerami (patrz poprzedni przykład z dwoma parami różniącymi się kolejnością położeń).

Dodatkowe założenia/uwagi

- Jeśli oczekiwana suma to X, to X nie pojawi się w żadnej z komórek tablicy.
- Rozwiązanie będzie jednoznaczne. Jeśli oczekiwana suma to np. 10 to nie pojawią się w tablicy 3 sąsiednie komórki o wartości 5.
- Pozycje startowe dla poszczególnych wątków będą od siebie odseparowane. Przykładowo: dla zaprezentowanego obrazka dwa wątki mogą uzyskać położenia: (2,3) i (6,2), gdzie pierwsza liczba to kolumna, druga to wiersz.
- Tablica będzie posiadać co najmniej jedną parę pozycji stanowiących rozwiązanie.
- Przed wykonaniem metody `start` wykonane zostaną metody `setThreadsFactory` oraz `setTable`.
- Fabryka wątków dostarczać będzie je w stanie nieuruchomionym. Metodę `start` należy wykonać samodzielnie.

Dostarczanie rozwiązania

Proszę o dostarczenie kodu **źródłowego** klasy `ParallelExplorer`. W klasie można umieścić własne metody i pola. Klasa ma implementować interfejs `Explorer`.

Plik z rozwiązaniem może zawierać inne klasy, ale tylko klasa `ParallelExplorer` może być publiczna.

Kodu, który sam dostarczam nie wolno modyfikować. W trakcie testów używany będzie w takiej formie, w jakiej został udostępniony. Proszę nie dołączać go do rozwiązań. Będzie dostępny w trakcie testów.

Programy będą testowane za pomocą czystego środowiska Java w wersji 21.

Linki

- Dokumentacja
- Kod źródłowy

Punktacja: do 0.5pkt. | Zadawanie pytań do: 2025-02-03 | Nadsyłanie rozwiązań do: 2025-02-05

Do zadania zgłoszono pytań: 8, a liczba odpowiedzi to: 8 | LINK do bazy pytań/odpowiedzi.

Możliwość dostarczania rozwiązań włączy się automatycznie po zakończeniu okresu zadawania pytań.

Zadanie 03 termin III. Do zdobycia maksymalnie: 1pkt.

Zadanie

Zadanie polega na użyciu OpenMP w celu przyspieszenia obliczeń. Kod to lekko zmodyfikowany program z zadania 1.

Na bazie `SequentialSwarm` należy wygenerować `ParallelSwarm`. Proszę zmienić SEQUENTIAL_SWARM_H_ na PARALLEL_SWARM_H_. Kod `ParallelSwarm` proszę poddać modyfikacjom za pomocą OpenMP.

Wymagania dla aplikacji równoległej

- Wersja równoległa i sekwencyjna muszą dawać ten sam wynik.
- Przyspieszyć należy **wszystkie** nadające się do tego celu metody `ParallelSwarm`
- Proszę nie zmieniać algorytmu/optymalizować pracy programu. Czas wykonania sekwencyjnej wersji oryginalnego kodu i po Państwa poprawkach ma być zbliżony. Poprawa szybkości pracy ma być związana **wyłączenie** ze zrównolegleniem kodu. Test: wykonanie oryginalnego kodu i Państwa kodu z jednym wątkiem (OMP_NUM_THREADS=1) - czas pracy ma być zbliżony.
- Dodatkowe wątki mają przyspieszać obliczenia!
- Proszę nie zmieniać w kodzie programu liczby tworzonych wątków. Ma działać ich dokładnie tyle, ile zostanie ustalone poprzez zmienną środowiskową OMP_NUM_THREADS.
- Program ma działać efektywnie - tj. dostając do dyspozycji kolejne rdzenie ma skracać czas potrzebny do uzyskania efektu.
- Kod musi dać się skompilować i uruchomić na udostępnianych komputerach Wydziałowych (klaster).

Kompilacja

Do skompilowania kodu powinno wystarczyć polecenie:

```
c++ -O2 -fopenmp *cpp
```

UWAGA: nagłówki omp.h i ParallelSwarm.h dodawane są jako wynik użycia przełącznika "-fopenmp".

Linki:

- Link do kodu źródłowego: - tutaj -

Dostarczanie rozwiązania:

- Rozwiązanie (klasa `ParallelSwarm`) ma rozszerzać (dziedziczyć) klasę `Swarm`. Idea użycia `Swarm` zapisana w main.cpp zostanie w trakcie testów zachowana.
- Klasa `ParallelSwarm` ma posiadać następujący konstruktor:

  ```
  ParallelSwarm( int robots, Antenna *antenna, Function *function  );
  ```

- Nagłówek o nazwie `ParallelSwarm.h` proszę również dostarczyć.
- Państwa kod musi dać się połączyć z oryginalnym kodem, który dostarczam dla tego zadania. Proszę nie zmieniać metod/konstruktorów w pozostałych klasach.
- Pliki .cpp i .h proszę wgrywać do systemu osobno.
- Uwaga: jeśli ktoś z Państwa chce, w ramach jednego terminu, dostarczyć więcej niż jedno rozwiązanie, to proszę zadbać o ponowne wgranie zarówno plików .cpp, jak i odpowiedniego pliku .h - nazwy tych plików muszą do siebie pasować. Czyli np. ParallelSwarm2.cpp i ParallelSwarm2.h Proszę jednak w "include" nadal używać ParallelSwarm.h (pliki otrzymają odpowiednie nazwy przed ich kompilacją).
- Warto zadbać o to, aby Państwa kod nie wyświetlał komunikatów na terminalu. Coś takiego może znacząco pogorszyć efektywności pracy programu.
- Proszę za pomocą "include" wskazywać wyłącznie pliki nagłówkowe. Uwaga dotyczy prób dołączania za pomocą "include" plików .cpp.

Testowanie

Udostępniam kod Main.cpp, który używany był w testach. W sumie nie robi on nic wielkiego - sprawdza poprawność wyników i szacuje efektywność pracy programu.

Link do kodu źródłowego: - tutaj -

Procedura testu

Z uwagi na użycie pliku z katalogu /proc do wyliczenia obciążenie CPU, test będzie działać pod Linux-em.

1. Do Main.cpp wpisujemy dane typu liczba robotów, liczb kroków itd. Parametry testu są o okolicy linii 600. Ja używam liczby robotów > 10000. I na tyle dużo kroków, że moja wersja

kodu pracuje około 400 sekund.
2. Kod kompilujemy z wybraną optymalizacją (np. -O3) ale **bez -fopenmp**
3. Program uruchamiamy i generuje on plik z danymi referencyjnymi "sequential.txt"
4. Kod kompilujemy z tą samą optymalizacją i z włączonym -fopenmp
5. Ponownie program uruchamiamy - generuje się podobny plik, ale na podstawie sekwencyjnego wykonania ParallelSwarm.
6. Teraz można już testować. Dla testów z liczbą wątków = 8 nie jest testowana efektywność. Dla innych liczb wątków działa pełen test.

Uwagi

- Są nieliczne programy, dla których program produkuje ujemny czas pracy. Dlaczego tak jest - nie wiem (tzn. nie wiem co zostało zrobione w ParallelSwarm, że coś takiego wychodzi).
- Efektywność > 100% jest zdecydowanie podejrzana. Też się zdarza... Innego wytłumaczenia jak istotna zmiana oryginalnego kodu na razie nie mam.

Zaliczenie

- Program musi zwracać poprawne wyniki (histogram, położenia robotów)
- Program musi osiągnąć co najmniej 50% średniej ważonej efektywności przy 7-miu rdzeniach (60% pochodzi z efektywności testu bez liczenia histogramów, 40% z oszacowanej efektywności liczenia histogramu).
- Punktacja:

```
                   I   II   III  IV
>= 50%           - 2.0  1.0  0.7  0.5
około 60%        - 2.6  1.6  0.8  0.6
około 62.5%      - 2.8  1.8  0.9  0.6
co najmniej 65% - 3.0  2.0  1.0  0.7
```

Czas trwania testów

Test jednego rozwiązania zajmuje u mnie około 12-15 minut.

Punktacja: do 1pkt. | Zadawanie pytań do: 2025-01-26 | Nadsyłanie rozwiązań do: 2025-01-29

Do zadania zgłoszono pytań: 2, a liczba odpowiedzi to: 2 | LINK do bazy pytań/odpowiedzi.

Autor: AA APlaceHolder | Kod: | Plik: Wybierz plik *Nie wybrano pliku* | Wyślij plik na serwer...