

OpenMP

Рачунарски системи високих перформанси

Горана Гојић¹ Вељко Петровић¹

¹Факултет техничких наука
Универзитет у Новом Саду

Рачунарске вежбе, Зимски семестар 2020/2021.

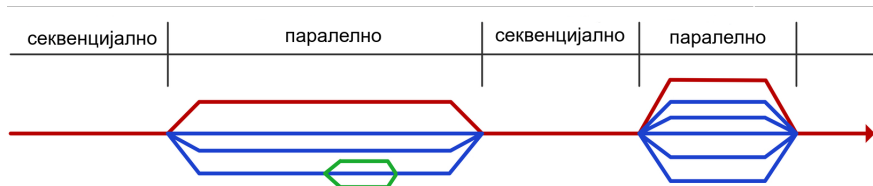


API за програмирање паралелних апликација на вишепроцесним (енг. *multiprocessing*) машинама, заснован на концепту дељења меморије.

Обухвата скуп компајлерских директива (енг. *compiler directives*), библиотечких рутина (енг. *library routines*) и променљивих окружења (енг. *environment variables*).

Постоји подршка за програмске језике C, C++ и Fortran.

fork-join модел



Паралелни регион (енг. *Parallel region*) - део програма који извршава тим нити

Мастер нит

Новокреиране нити - идентификатори од 1 до $N - 1$, не постоје након паралелног региона у којем су креиране
Све нити у паралелном региону чине **тим нити**.

¹Оригинална илустрација преузета из књиге *Parallel Computing Book*    

У OpenMP се тим нити креира коришћењем `parallel` конструкције (енг. *parallel construct*)

```
#pragma omp parallel [klauzule]  
    strukturirani-blok
```

Структурирани блок:

- Има тачно једну улазну тачку
- Има тачну једну излазну тачку (не може садржати `break`, `goto`, може `exit`)

Клаузуле: `num_threads`, ...

Пример 1: Добри стари Hello World!

```
#include <stdio.h>
#include <omp.h>

int main() {

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf("Hello(%d)", id);
        printf(" world!(%d)\n", id);
    }

    return 0;
}
```

Пример 1: добри Hello Стари World!

Један пример извршавања:

```
Hello(1)Hello(2) world!(2)
```

```
Hello(0) world!(0)
```

```
world!(1)
```

```
Hello(3) world!(3)
```

Иако нигде у примеру број нити није задат, OpenMP радно окружење (енг. *execution environment*) је одлучило да направи 4 нити.

Типично, OpenMP радно окружење прави онолико нити колико има језгара (физичких или логичких). Нпр. овај испис је добијен извршавањем на Интеловом процесору i5 3337U (4 логичка језгра).

Пример 1: добри Hello Стари World!

Како створити неки произвољан број нити у паралелном региону?

```
omp_set_num_threads()  OMP_NUM_THREADS
```

Могуће је и да радно окружење **не направи** онолико нити колико сте од њега затражили услед постојећих ограничења!

Друге интересантне функције: `omp_get_num_threads()`,
`omp_get_thread_num()`, `omp_get_max_threads()`

Превођење изворне датотеке

Како компајлирати? Отворити терминал, позиционирати се у директоријум у којем се налази изворна датотека (`izvornad.c`) и унети:

```
gcc [-g] [-o izvrsnad] izvornad.c -fopenmp [-O2]
```

У случају да постоји више изворних датотека, потребно их је све навести. Делови у углатим заградама нису обавезни.

Опције:

- `-g` - омогућава прикупљање података за дебаговање
- `-o` - специфицира назив излазне датотеке, у овом случају то је извршна датотека
- `-O2` | `-O3` - индикатор компајлеру да примени одређене скупове оптимизација (`O3` већи скуп оптимизација од `O2`). Употреба ових опција се **не препоручује** у комбинацији са `-g` опцијом!

Условно превођење

У одређеним случајевима је могуће **исти** OpenMP изворни код превести и у секвенцијални и у паралелни програм.

```
#ifdef _OPENMP  
// pozivi OpenMP api-ja  
#endif
```

Преводац игнорише непознате директиве.

Једно решење које ради као паралелно на платформама са OpenMP подршком, а на осталим ради као секвенцијално.

Како покренути искомпајлирано решење? Отворити терминал, позиционирати се у директоријум у којем се налази извршна датотека (izvrsnad) и унети:

```
./izvrsnad <lista-argumenata>
```

У случају да при компајлирању није наведено име извршне датотеке, она ће се подразумевано звати a.out.

Задатак 1: Рачунање вредности броја π

Коришћењем само `parallel` конструкције, паралелизовати програм који рачуна вредност интеграла

$$\int_0^1 \frac{4}{(1+x^2)}$$

Секвенцијална имплементација програма у `C` програмском језику је дата у директоријуму `zadaci`. Резултат интеграљења би требало да буде једнак броју π . Потребно је додати `parallel` конструкцију без даљих модификација секвенцијалног програма.

Шта се дешава са резултатом?

Пример решења: функција `parallel_code_incorrect`, датотека `pi.c`, директоријум `resenja`.

Опсег видљивости променљивих

- Дељене
 - Декларисане изван паралелног региона (нпр. `sum` из задатка 1)
 - Све OpenMP нити унутар региона имају приступ истој променљивој - **трка до података**
- Приватне
 - Свака OpenMP нит има своју инстанцу променљиве
 - Декларисане унутар паралелног региона
 - Бројачка променљива `for` петље првог нивоа
 - Променљиве декларисане у функцији позваној из паралелног региона

Задатак 2: Рачунање вредности броја π

Модификовати решење претходног задатка тако да се уклони штетно преплитање.

Пример решења: функција `parallel_code`, датотека `pi.c`, директоријум `resenja`.

Задатак 3: Рачунање вредности броја π

Паралелно решење задатка 2 елиминише проблем штетног преплитања, али уводи проблем лажног дељења (енг. *false sharing*) при приступу низовној променљивој `sum`. Изменити решење тако се отклони лажно дељење.

Напомена: размислити о томе колико се елемената низа преноси у кеш процесора када се приступа једном елементу.

Пример решења: функција `parallel_code_no_false_sharing`, датотека, `pi.c`, директоријум `resenja`.

Конструкције за експлицитну синхронизацију

Синхронизација високог нивоа апстракције:

- `barrier construct` - дефинише тачку у коду до које све активне нити морају да се зауставе док до те тачке не стигне и последња нит, након чега све нити могу наставити даље извршавање.

`#pragma omp barrier`

- `critical construct` - само једна нит може у једном тренутку бити у критичној секцији.

`#pragma omp critical`
strukturirani-blok

- `atomic construct` - хардверски подржан искључив приступ ажурирању вредности просте променљиве. Уколико нема хардверске подршке, ова конструкција се понаша као и `critical`.

`#pragma omp atomic`

Задатак 4: Рачунање вредности броја π

Дорадити решење задатка 2 тако да се штетно преплитање уклони одговарајућим синхронизационим механизмом.

Пример решења: функција `parallel_code_synchronization`, датотека `pi.c`, директоријум `resenja`.

Конструкције за поделу посла

енг. *worksharing constructs*

- loop конструкција (енг. *loop construct*)
- sections/section конструкција (енг. *sections/section construct*)
- single конструкција (енг. *single construct*)

На крају блока кода који се извршава у склопу неке од конструкција за поделу посла подразумевано постоји **имплицитна баријерна синхронизација**.

Подразумевано понашање се може променити навођењем `nowait` клаузе у оквиру директива за поделу посла (пример касније).

Имплицитна баријерна синхронизација

Имплицитна баријерна синхронизација се такође подразумевано налази и на **крају паралелног региона**, али је за разлику од конструкција за поделу посла, **није могуће одатле уклонити**.

Зашто?

loop конструкција

Синтакса:

```
#pragma omp for [klauzule]  
for-petlje
```

Проблеми у рачунарству високих перформанси се често свODE на рад са великим низовима! То значи да често постоји итерирање кроз низове...

Тај посао може да се подели на више нити! Свака нит у паралели може обрадити парче низа.

Овај начин расподеле података сте већ имплементирали у задатку 2, али сте сами морали да специфицирате границе низа над којима ради свака појединачна нит.

loop конструкција

Синтакса:

```
#pragma omp for [klauzule]  
for-petlje
```

Клаузуле:

- schedule
- collapse
- private
- shared
- reduction
- nowait
- ...

Пример 2: loop конструкција

Пример употребе **for** директиве

```
#pragma omp parallel
{
    int sum = 0;
    #pragma omp for
    for (int i = 0; i < N; i++)
        sum += A[i];
}
```

Коришћењем комбиноване конструкције:

```
#pragma omp parallel for
{
    int sum = 0;
    for (int i = 0; i < N; i++)
        sum += A[i];
}
```

loop конструкција: распоређивање

Како ће итерације петље бити додељене нитима?

OpenMP подржава више стратегија распоређивања које се могу специфицирати `schedule` клаузом.

```
#pragma omp for schedule(tip [,velicina_bloka])
```

loop конструкција: распоређивање

Како ће итерације петље бити додељене нитима?

- `static` - блокови итерација се додељују нитима у време компајлирања по `round-robin` принципу
- `dynamic` - блокови итерација се додељују нитима у време извршавања тако да оптерећење нити буде оптимално
- `guided` - модификација динамичког распоређивања где је сваки наредни блок додељен нити мањи од претходног
- `auto` - компајлер одређује тип распоређивања који мисли да је најбољи за проблем
- `runtime` - могуће је "споља" утицати на тип распоређивања преко `OMP_SCHEDULE` променљиве окружења

Присетимо се непотпуног паралелног сабирања елемената низа из примера 2

```
#pragma omp parallel for
{
    int sum = 0;
    for (int i = 0; i < N; i++) {
        sum += A[i];
    }
}

/* saberi parcijalne sume */
```

Да би програм био потпуно функционалан, потребно је посабирати парцијалне суме које срачунају нити.

Може да се уради овако:

```
int sum = 0;
#pragma omp parallel for
{
    for (int i = 0; i < N; i++) {
        #pragma omp critical
        sum += A[i];
    }
}
```

И у дељеној променљивој `sum` ће бити коначан резултат. Али ово је **ВЕОМА** неефикасно!

А може и овако:

```
int sum = 0;
#pragma omp parallel for reduction(+:sum)
{
    for (int i = 0; i < N; i++) {
        sum += A[i];
    }
}
```

Шта заправо значи `reduction(+:sum)`?

- 1 За сваку нит у паралелном региону направи по једну приватну инстанцу променљиве `sum` и иницијализуј је на вредност неутралну за наведени редукциони оператор (у случају сабирања је то 0).
- 2 Свака нит мења своју копију променљиве `sum`.
- 3 На крају петље, резултати се комбинују употребом редукционог оператора у дељену променљиву `sum`.

Редукције

Општи формат редукције:

```
reduction(редукциони_operator : lista_promenljivih)
```

Уграђени редукциони оператори за C/C++:

+	0
*	1
-	0
min	највећи позитивни број
max	највећи негативни број
&	~ 0
	0
^	0
&&	1
	0

Задатак 5: Рачунање вредности броја π

Имплементирати паралелно решење рачунања вредности броја π уз коришћење `for` директиве и `reduction` клаузе.

Пример решења: функција `parallel_code_for_construct`, датотека `pi.c`, директоријум `resenja`.

Имплицитна vs. експлицитна баријера

Експлицитна баријера је задата *#pragma omp barrier* директивом.

Имплицитна баријера је баријера већ укључена у неку конструкцију (нпр. *for* конструкцију).

Имплицитна vs. експлицитна баријера

У којем делу кода ће се нити синхронизовати?

```
#pragma omp parallel  
{  
    /* први блок наредби */  
    #pragma omp barrier  
    /* други блок наредби */  
}
```

Имплицитна vs. експлицитна баријера

У којем делу кода ће се нити синхронизовати?

```
#pragma omp parallel  
{  
    /* први блок наредби */  
    #pragma omp barrier  
    /* други блок наредби */  
}
```

Одговор: На `barrier` директиви (експлицитна баријерна синхронизација).

Имплицитна vs. експлицитна баријера

У којем делу кода ће се нити синхронизовати?

```
#pragma omp parallel for  
for (i = 0; i < N; i++)  
    /* prvi blok naredbi */  
/* drugi blok naredbi */
```

Имплицитна vs. експлицитна баријера

У којем делу кода ће се нити синхронизовати?

```
#pragma omp parallel for  
for (i = 0; i < N; i++)  
    /* први блок наредби */  
    /* други блок наредби */
```

Одговор: На крају првог блока наредби. Све нити морају да заврше своје итерације петље да би могле да наставе са другим блоком наредби, јер подразумевано `loop` конструкција има уграђену имплицитну баријеру.

Имплицитна vs. експлицитна баријера

У којем делу кода ће се нити синхронизовати?

```
#pragma omp parallel for nowait  
for (i = 0; i < N; i++)  
    /* prvi blok naredbi */  
/* drugi blok naredbi */
```

Имплицитна vs. експлицитна баријера

У којем делу кода ће се нити синхронизовати?

```
#pragma omp parallel for nowait  
for (i = 0; i < N; i++)  
    /* prvi blok naredbi */  
    /* drugi blok naredbi */
```

Одговор: На крају паралелног региона. Имплицитна баријера `loop` конструкције је онемогућена употребом `nowait` клаузе.

sections/section конструкција

Свака нит унутар `sections` конструкције извршава један блок кода који припада секцији.

Синтакса:

```
#pragma omp sections [klauzule]
{
    [ #pragma omp section ]
        strukturirani-blok
    [ #pragma omp section
        strukturirani-blok ]
    ...
}
```

Пример 3: sections/section конструкција

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
            x_calculation();
        #pragma omp section
            y_calculation();
        #pragma omp section
            z_calculation();
    }
}
```

single конструкција

Синтакса:

```
#pragma omp single [klauzule]  
strukturirani-blok
```

Блок наредби извршава само нит која прва уђе у структурирани блок.

```
#pragma omp parallel  
{  
    do_many_things();  
    #pragma omp single  
        { exchange_boundaries(); }  
    #pragma omp barrier  
        do_many_other_things();  
}
```

master конструкција

Синтакса:

```
#pragma omp master  
    strukturirani-blok
```

Блок наредби извршава само мастер нит.

```
#pragma omp parallel  
{  
    do_many_things();  
    #pragma omp master  
        { exchange_boundaries(); }  
    #pragma omp barrier  
        do_many_other_things();  
}
```

Нема имплицитне синхронизације.

Синхронизациони механизми: `lock`

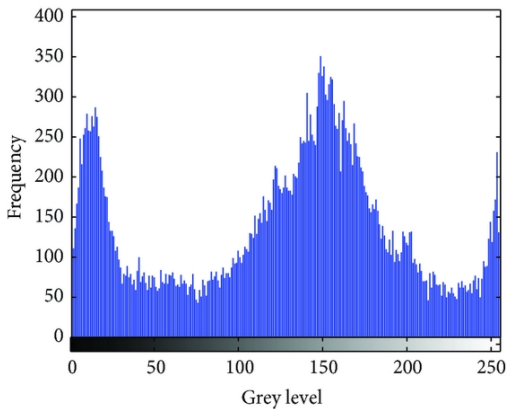
Припада механизмима ниског нивоа синхронизације.

Аналогни појам пропусници из C++11 вишенитног окружења.

`critical` директива у позадини користи `lock`. Зашто бисте онда икада желели да директно користите ову директиву?

Нема проблема, ако на пример, ограничавате приступ једној целобројној променљивој кроз критичну секцију. Али шта ће се десити ако кроз критичну секцију ограничите приступ елементима неког низа?

Пример 4: Гистограм



```
int histogram[255];
```

Пример 4: Гистограм

```
#pragma omp parallel for
for (i = 0; i < NBUCKETS; i+) {
    omp_init_lock(&hist_locks[i]);
    hist[i] = 0;
}

#pragma omp parallel for
for (i = 0; i < NVALS; i++) {
    ival = (int) sample(arr[i]);
    omp_set_lock(&hist_locks[ival]);
    hist[ival]++;
    omp_unset_lock(&hist_locks[ival]);
}

for (i = 0; i < NBUCKETS; i++) {
    omp_destroy_lock(&hist_locks[i]);
}
```

Клаузе за рад са подацима

- `shared(<lista-promenljivih>)`
- `private(<lista-promenljivih>)`
- `firstprivate(<lista-promenljivih>)`
- `lastprivate(<lista-promenljivih>)`
- `default(private | shared | none)` - Ако се ништа не наведе за ову клаузу, подразумевана вредност у C и C++ програмским језицима је `shared`.

Пример 5: Клаузе за рад са подацима

```
void dummy() {  
    int tmp = 0;  
    #pragma omp parallel for private(tmp)  
    for (int j = 0; j < 5; j++)  
        tmp += j;  
    printf("%d\n", tmp);  
}
```

Која вредност ће бити исписана на стандардни излаз?

Пример 6: Клаузе за рад са подацима

```
void dummy() {  
    int tmp = 0;  
    #pragma omp parallel for private(tmp)  
    for (int j = 0; j < 5; j++)  
        tmp += j;  
    printf("%d\n", tmp);  
}
```

Која вредност ће бити исписана на стандардни излаз?

Одговор: На стандардни излаз ће бити исписана вредност 0.

Објашњење: Како је променљива `tmp` проглашена приватном, свака нит ће имати своју инстанцу ове променљиве. По завршетку петље, локалне променљиве ће бити уништене, а дељена променљива `tmp` ће задржати свој у иницијалну вредност.

Пример 6: Клаузе за рад са подацима

```
void dummy() {  
    int tmp = 0;  
    #pragma omp parallel for private(tmp)  
    for (int j = 0; j < 5; j++)  
        tmp += j;  
    printf("%d\n", tmp);  
}
```

Која је иницијална вредност приватних верзија променљиве tmp?

Пример 6: Клаузе за рад са подацима

```
void dummy() {  
    int tmp = 0;  
    #pragma omp parallel for private(tmp)  
    for (int j = 0; j < 5; j++)  
        tmp += j;  
    printf("%d\n", tmp);  
}
```

Која је иницијална вредност приватних верзија променљиве tmp?

Одговор: Иницијална вредност приватних променљивих tmp је непозната.

Објашњење: Клауза `private(tmp)` каже компајлеру да треба да алоцира простор за променљиву tmp на стеку. Компајлер не мора иницијализовати заузету локацију.

Пример 6: Клаузе за рад са подацима

```
void dummy() {  
    int tmp = 0;  
    #pragma omp parallel for private(tmp)  
    for (int j = 0; j < 5; j++)  
        tmp += j;  
    printf("%d\n", tmp);  
}
```

Којом клаузом је потребно заменити `private` клаузу да би локалне инстанце променљиве `tmp` добиле иницијалну вредност глобалне променљиве `tmp`?

Пример 6: Клаузе за рад са подацима

```
void dummy() {  
    int tmp = 0;  
    #pragma omp parallel for private(tmp)  
    for (int j = 0; j < 5; j++)  
        tmp += j;  
    printf("%d\n", tmp);  
}
```

Којом клаузом је потребно заменити `private` клаузу да би локалне инстанце променљиве `tmp` добиле иницијалну вредност глобалне променљиве `tmp`?

Одговор: `firstprivate`

Задатак 6: Манделбровов сет

Дата је датотека `mandelbrot.c`. Датотека садржи паралелно OpenMP решење које одређује Манделбровов сет. Решење има пар проблема везаних за коришћење клауза за рад са подацима и понеко штетно преплитање. Пронађите и исправите грешке.

Решење: датотека `mandelbrot.c`, директоријум `resenja`

Како паралелизовати `while` и рекурзију?

Иницијално, OpenMP је замишљен тако да је могуће паралелизовати проблеме за које се унапред зна број потребних итерација за њихово решавање!

Проблем! Како применити OpenMP у другим типовима петљи у којима се не зна унапред број итерација? Или у случају рекурзије?

Како паралелизовати `while` и рекурзију?

Иницијално, OpenMP је замишљен тако да је могуће паралелизовати проблеме за које се унапред зна број потребних итерација за њихово решавање!

Проблем! Како применити OpenMP у другим типовима петљи у којима се не зна унапред број итерација? Или у случају рекурзије?

Опције:

- Проблем трансформисати у форму `for` петље ако је то могуће
- Користити `task` конструкцију (од OpenMP 3.0)

Пример 7: Паралелизација обраде чворова листе

```
// 1. odrediti broj cvorova
while (p != NULL) {
    nelems++; p = p->next;
}
...
// 2. zapamtiti adrese cvorova
p = head;
for (i = 0; i < nelems; i++) {
    ptrs[i] = p;
    p = p->next;
}
...
// 3. pokrenuti paralelnu obradu
#pragma omp parallel for
for (i = 0; i < nelems; i++)
    processwork(ptrs[i]);
```

```
// petlja koju
// treba paralelizovati
while (p != NULL) {
    processwork(p);
    p = p->next;
}
```

task конструкција

Синтакса:

```
#pragma omp task [klausule]  
strukturirani-blok
```

Може се посматрати као независна јединица посла.

Чине је:

- Код који задатак извршава
- Подаци (приватне и дељене променљиве)
- Internal Control Variables (ICV) (нпр. индикатор да ли задатак може да буде додељен различитим нитима, врста распоређивања, број нити у паралелном региону итд.)

Пример 8: Креирање задатака

`task` и `single` конструкције се често користе заједно: једна нит прави задатке који се увезују у ред задатака одакле све нити могу да узимају задатке.

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        y = f(x)
        #pragma omp task
        z = g(x)
    }
}
```


Задатак 7: Модификација листе

Дата је секвенцијална имплементација листе (`linked.c`) у којој сваки елемент садржи по један Фибоначијев број добијен функцијом `processwork`. Направити OpenMP паралелни програм коришћењем `task` конструкције.

Решење: датотека `linkedlist.c`, директоријум `resenja`.

Конструкције за синхронизацију извршавања задатака

- `taskwait` - синхронизација само задатака истог нивоа

```
#pragma omp taskwait
```

- `taskgroup` - синхронизује и подзадатке

```
#pragma omp taskgroup  
strukturirani-block
```

- `depend` - task клаузула

```
depend(in | out | inout : <lista-promenljivih>)
```

Пример 9: Синхронизација зависних задатака

```
int y, z;  
#pragma omp parallel  
{  
    #pragma omp single  
    {  
        #pragma omp task  
        y = f(x)  
        #pragma omp taskwait  
        #pragma omp task  
        z = g(y)  
    }  
}
```

Пример 9: Синхронизација зависних задатака

```
#pragma omp parallel
{
    int y, z;
    #pragma omp single
    {
        #pragma omp task depend(out:y)
        y = f(x)
        #pragma omp task depend(in:y)
        z = g(y)
    }
}
```

Први задатак модификује y (**out**: y), а други задатак чита y (**in**: y). Други задатак мора бити распоређен пре првог.

Задатак 8: Множење матрице и вектора

Имплементирати секвенцијални програм за множење неквадратне матрице и вектора у С програмском језику.

Након што се уверите да програм даје очекиване резултате, имплементирани OpenMP паралелни алгоритам на основу секвенцијалног програма.

Напомене:

- Свака од димензија матрице треба да буде макар 1000. За потребе тестирања програма димензије матрице могу да буду и мање.
- Мерити извршавање програма функцијом `omp_get_wtime()`.

Задатак 9: Множење матрица - **домаћи**

Имплементирати секвенцијални програм за множење две неквадратне матрице у С програмском језику.

Након што се уверите да програм даје очекиване резултате, имплементирати OpenMP паралелни програм на основу секвенцијалног програма.

Напомене:

- Коректан секвенцијални програм тестирати на великим матрицама (око 1000 по димензији, модификовати зависно од карактеристика рачунара на којем радите задатак). За потребе тестирања, димензије матрица могу да буду и мање.
- Мерити извршавање програма функцијом `omp_get_wtime()`.

Задатак 10: Акумулирање вредности чворова стабла

Направити секвенцијалну имплементацију стабла у С програмском језику. Сваки чвор стабла садржи један разломљени број у једнострукој прецизности. Потребно је имплементирати минимални скуп функционалности (креирање стабла, сабирање вредности чворова стабла и уништавање стабла).

Након што се уверите да програм даје очекиване резултате имплементирати OpenMP паралелни програм на основу секвенцијалног програма.

Напомене:

- Паралелни програм имплементирати коришћењем `task` конструкције.
- Мерити извршавање програма функцијом `omp_get_wtime()`.

Задатак 11: Транспонованње матрице

Дата је секвенцијална имплементација транспонованња матрице (директоријум `zadaci/matrix_transpose`). Направити OpenMP паралелну верзију алгоритма.

Напомене:

- Мерити извршавање програма функцијом `omp_get_wtime()`.

Задатак 12: Једноставни генетски алгоритам

Дата је секвенцијална имплементација једноставног генетског алгоритма имплементираног у С програмском језику (директоријум `zadaci/genetic_algorithm`). Покренути секвенцијални алгоритам над свим датим примерима према упутству у `README.md` датотеци и анализирати времена извршавања делова генетског алгоритма.

Одредити критичне делове кода и паралелизовати их коришћењем OpenMP.

Напомене:

- Мерити извршавање програма функцијом `omp_get_wtime()`.

Задатак 13: Тражење корена функције над интервалом - домаћи

Дата је секвенцијална имплементација методе за одређивање корена функције над задатим интервалом методом бисекције (директоријум `zadaci/bisection`). Покренути секвенцијални алгоритам над свим датим примерима према упутству у `README.md` датотеци и погледати решења сва три задата примера.

Затим имплементирати OpenMP паралелно решење.

Напомене:

- Алгоритам испробати на још примера. При провери тачности добијеног решења могуће је користити неки од алата за одређивање корена функције (Волфрам, на пример).
- Мерити извршавање програма функцијом `omp_get_wtime()`.

- Документација са OpenMP сајта, видети www.openmp.org
- "Introduction to OpenMP", Tim Mattson, доступно на овом линку
- "Introduction to OpenMP", пратећа презентација
- "Parallel Computing Book", Victor Eijkhout, електронска верзија књиге је доступна на овом линку