Тема вежбе: Туторијал – Паралелно програмирање

ПАРАЛЕЛНО ПРОГРАМИРАЊЕ – *ОПЕН МП*

Дељена меморија - концепт

Савремени вишепроцесорски рачунарски системи се углавном састоје од више процесорских јединица и заједничке меморије. Овакви системи са дељеном меморијом имају јединствен адресни простор у целом меморијском систему, где сваки процесор може равномерно да приступа свим меморијским локацијама у систему.

Програмски модел дељене меморије се заснива на коришћењу нити, које могу приступати дељеним подацима. Како би паралелни програми били што боље употребљени, неопходна је размена података између нити, које комуницирају преко уписа и читања дељених података. Нпр. НИТ 1 уписује вредност у дељену променљиву А, а затим НИТ 2 може да чита вредност из А. Свака нит извршава програмске инструкције независно од осталих нити и неопходно је обезбедити коректно извршавање акција над дељеним променљивама.

Паралелни задатак (енгл. *Task*) је део израчунавања, који је могуће извршити независно од других задатака. За извршавање сваког задатка је могуће креирати нову нит, али за велики број мањих послова ово може бити веома „скупо“. Уместо тога, задаци се могу извршавати помоћу унапред припремљеног скупа нити (енгл. *Thread pool*). У овом случају, сваки од задатака се распоређује на неку од нити из скупа, у оквиру које се извршава.

Петље су често најпогоднија места за увођење паралелизма у програмима. Уколико итерације у петљи немају међусобну зависност података, могуће их је расподелити различитим нитима и такве итерације се могу извршавати у било ком редоследу. Могуће је паралелизовати само оне петље код којих се број итерација може унапред израчунати (добро формиране петље).

Уколико имамо петљу као у следећем примеру и две нити, могуће је итерирати део од 0 до 49 једној, а део 50-99 другој нити:

```
for (i = 0; i < 100; i++)
{
    a[i] += b[i];
}
```

Директан рад са нитима, користећи уграђену подршку у језик (Ц++ нити) или системске библиотеке (нпр. Посикс нити) су погодне за вишепроцесорско програмирање на ниском нивоу, али превише општи и компликовани за инжењерске примене.

ОпенМП

ОпенМП је једноставана програмска спрега (енг. *Application programming interface, API*) дизајнирана за програмирање паралелних рачунарских система са дељеном меморијом, који је заснован на концепту нити и паралелних задатака. OpenMP није програмски језик, већ скуп надоградњи за програмске језике *Фортран*, *Ц* і *Ц++*. Већина савремених компајлера поседује уграђену подршку за ОпенМП, што значајно олакшава његово коришћење. Да би се успешно превео програмски код који користи *ОпенМП*, довољно је само компајлеру проследити одговарајуће опције приликом превођења:

- **`-fopenmp`** за ГЦЦ
- **`-openmp`** за Интел и Оракл преводиоце
- **`/openmp`** за Мајкрософт преводиоце

ОпенМП надоградње се састоје из:

- Компајлерских директива (енгл. *Compiler directives*)
- Библиотеке са помоћним функцијама (енгл. *Runtime library routines*)
- Променљивих окружења (енгл. *Environment variables*)

ОпенМП значајно поједностављује програмирање, јер је у многим случајевима додавање једне линије кода довољно за омогућавање паралелног извршавања.

Компајлерске директиве и ознаке:

Компајлерске директиве у општем случају представљајупосебне линије изворног кода, које служе за укључивање или искључивање одређених подешавања компајлера, и имају значење само преводиоцима који их подржавају (нису дефинисане стандардом језика). Преводиоци који их не подржавају, једноставно их игноришу. Компајлерске директиве започињу су увек у формату „#pragma *tekstdirektive*”. Све директиве које су део ОпенМП-а започињу са ознаком *omp:*

```
#pragma omp
```

ОпенМП директиве се игноришу ако се код преводи као обичан секвенцијални код (уколико компајелру није прослеђена одговарајуће опција). Свака директива се односи на прву наредну линију у коду:

```
#pragma omp parallel
    jedan iskaz (do sledećeg ';')
```

Или на блок кода обухваћен **{ },** који следи одмах након директиве.

```
#pragma omp parallel
{
    blok koda
}
```

Библиотеке са помоћним функцијама и променљиве окружења:

Помоћне функције и променљиве окружења служе за конфигурацију окружења ОпенМП и добављање информација о тренутним подешавањима. Све помоћне функције и променљиве окружења декларисане су у заглављу *omp.h.*

Пример једне променљиве окружења јесте број нити који ће се користити у току извршавања: ***OMP_NUM_THREADS***. Ова вредност се може поставити из оперативног система или одговарајућим помоћним функцијама **.**

Неке од помоћних функција које нуди ОпенМП су дате у Табели 1.1.

Функција	Шта ради?
`int omp_get_num_threads();`	Дивавља број тренутно коришћених нити. Враћа 1 ако се зове изван паралелног региона.
`int omp_get_thread_num();`	Добавља редни број нити која се тренутно извршава. Узима вредности од `0` до `omp_get_num_threads() - 1`.
`void omp_set_num_threads(int n);`	Поставља жељени број нити који ће бити коришћен.
`int omp_in_parallel();`	Враћа 1 уколико је позвана унутар паралелног региона, 0 уколико је ван њега.
`int omp_get_max_threads();`	Враћа колико је укупно нити доступно за коришћење (максималан број нити за било који паралелни регион)
`double omp_get_wtime();`	Враћа време у секундама, протекло од одређеног тренутка у прошлости. Тренутак није прецизно дефинисан, па је потребно користити вредности искључиво за израчунавање временског интервала.

Табела 1-1: Основне функције ОПЕН МП библиотеке

У наставку је дат једноставан пример програмског кода у програмском језику Ц, који користи ОпенМП директиве. Покретање програма се обавља на исти начин као и за обичан секвенцијални код.

```
#include <iostream>
#include <omp.h>
using namespace std;

int main()
{
    printf("Starting off in the sequential world.\n");
    #pragma omp parallel
    {
        cout << "Thread number" << omp_get_thread_num() << endl;
    }

    cout << "Back to the sequential world." << endl;
    return 0;
}
```

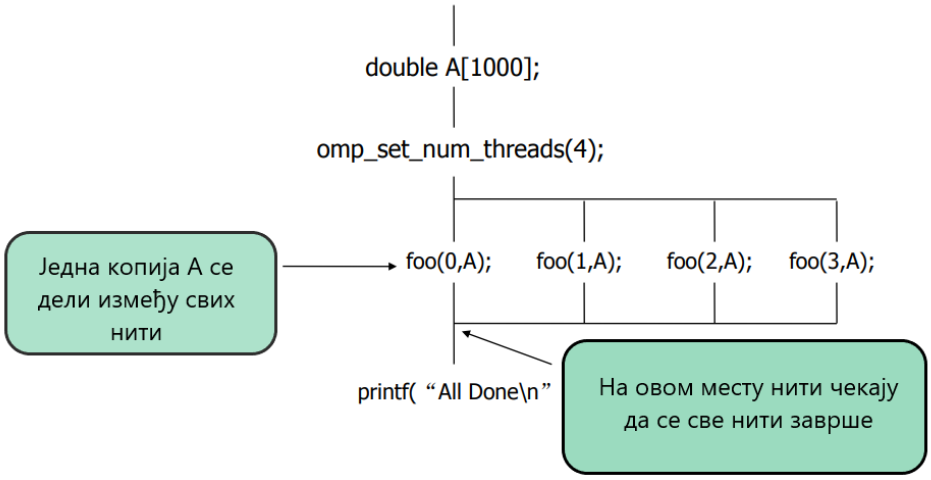
Паралелни региони:

Паралелни регион је основна паралелна конструкција у ОпенМП-у. Сваки програм почиње извршавање у једној (главној) нити која започиње извршавање функције *main*. Када се наиђе на први паралелни регион (директиву која га означава), главна нит креира скуп нити. Свака нит у скупу има свој идентификациони број (ИБ), који има целобројну вредност. Главна нит има ИБ = 0. Свака нит извршава наредбе задате унутар паралелног региона. Код који се налази унутар региона ће бити извршен онолико пута колико има доступних нити. Главна нит равноправно учествује у послу и на крају паралелног региона чека на остале нити да заврше посао, а затим наставља са извршавањем осталих наредби.

Овај начин рада назива се шаблон „измрести-прикључи“ (енг. *fork-join*) и основни је модел паралелног извршавања у оквиру ОпенМП-а.

Паралелни региони препуштају програмеру одлуку о подели посла. Нпр. да би се креирала четири паралелна региона у следећем примеру, свака нит позива функцију *foo* са идентификатором 0 до 3. Свака нит извршава код унутар структуираног блока. Након што све 4 нити заврше посао, програм наставља са даљим секвенцијалним извршавањем кода, односно исписом „*All Done*”.

```
    double A[1000];
    omp_set_num_threads(4);

    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        foo(ID, A);
    }
    cout << "All Done" << endl;
```



Слика 1 - Паралелни региони

Паралелна петља:

Паралелни региони представљају најпростију конструкцију у оквиру ОпенМП. Недостатак овог приступа је то што корисник мора ручно да имплементира поделу посла, синхронизацију између послова, и слично. Због тога се много чешће користе шаблони вишег нивоа које ОпенМП нуди. У наставку следи опис једног од њих.

Најчешћи начин за паралелизацију рачунских операције на вишејезгарном процесору је паралелизација *for* петље. ОпенМП нуди шаблон којим се на прост начин то може урадити користећи директиву за паралелну петљу. Пре него што употребимо овакву конструкцију, морамо бити сигурни да је свака итерација петље независна једна од друге, како би се заиста могле извршавати у паралели.

У општем случају, не може се предвидети која нит ће преузети извршавање одређене итерације. ОпенМП посматра индекс петље као приватан. Уколико има више угњеждених петљи, директива за паралелну петљу се односи само на спољашњу. Могуће је увести паралелизам и у унутрашње петље, додавањем директива за сваку петљу посебно. Уколико се петља већ налази у паралелном региону, директива за паралелизацију петље је:

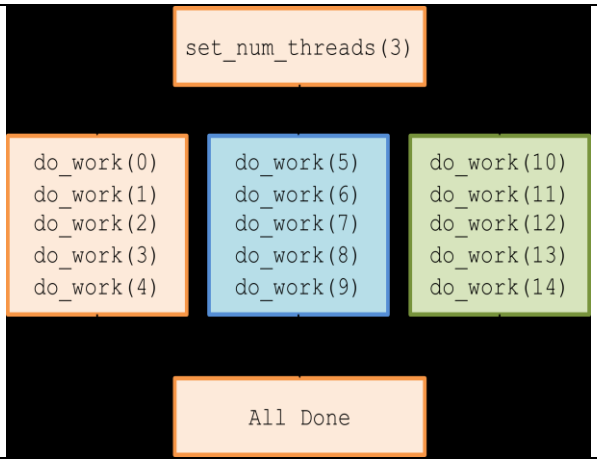
```
#pragma omp for
```

Овом директивом се не прави посебан тим нити спремних за преузимање задатака, већ се подразумева да већ постоје. Уколико се налазимо изван паралелног региона, могуће је у оквиру једне директиве направити регион и назначити да читав регион представља петљу. У оба случаја прва инструкција након директиве мора бити *for* петља.

```
#pragma omp parallel for
```

У наставку је дат пример једне паралелне петље и графички приказ њеног извршавања. Наранџастом бојом је приказано оно што извршава нит 0, плавом оно што извршава нит 1, а зеленом оно што извршава нит 2.

```
omp_set_num_threads(3);

#pragma omp parallel for
for (int i = 0; i < 15; i++)
{
    do_work(i);
}

cout << "All Done" << endl;
```



Наредни пример приказује попуњавање вредности вектора користећи паралелну петљу. У примеру је приказано и на који начин је могуће измерити време извршавања паралелне петље. Када мерите време, неопходно је превести пројекат користећи конфигурацију ***Release***.:

```
template <typename T>
void print_vector(T &vec) // Funkcija za ispis sadržaja vektora
{
    cout << "{ ";
    for (T::iterator it = vec.begin(); it < vec.end(); it++)
        cout << *it << " ";
    cout << "}" << endl;
}

int main(int argc, char **argv)
{
    const int n = 10;
    vector<int> x(n), y(n);

    float startTime = omp_get_wtime(); // Dobavi trenutno vreme (sec)

    #pragma omp parallel for
    for (int i = 0; i < n; ++i)
    {
        x[i] = i;
        y[i] = 2 * i;
        do_work(i);
    }

    float endTime = omp_get_wtime(); // Dobavi vreme nakon petlje
```

```
    cout << "Run time: " << (endTime - startTime) * 1000 << "ms." <<
endl;   // Ispis dužine trajanja izvršavanja petlje u milisekundama
    print_vector(x); // Ipis sadržaja prvog vektora
    print_vector(y); // Ipis sadržaja drugog vektora

    return 0;
}
```

Одредбе за подешавање директива:

За специфицирање додатних информација у директиви за паралелни регион, користе се одредбе илити клаузуле (енг. *clauses*). Одредбе се одвајају празним местом од директиве.

```
    #pragma omp parallel [clauses]
```

Дељене и приватне променљиве:

У оквиру паралелних региона, променљиве могу бити **дељене** (*shared*) или **приватне** (*private*). Код дељених променљивих све нити виде исту копију променљиве и могу да читају или пишу у њих. За разлику од дељених, код приватних променљивих свака нит има своју копију приватне променљиве, која је невидљива за остале нити (слично прослеђивању параметара функцији по вредности). Из приватне променљиве може да чита или да у њу пише само нит која је њен власник.

Све променљиве које су дефинисане унутар паралеллног региона су **увек приватне** за сваку нит. Све променљиве које су дефинисане пре паралелног региона, а видљиве су унутар паралелног региона су **подразумевано дељене**. У претходном примеру, променљива A је дељена између све 4 нити, док је променљива ID приватна.

То да ли ће нека променљива, која постоји и ван региона, дељена или приватна може се подешавати користећи неку од две наредне одредбе у наставку директиве:

`shared (var_list)` - све променљиве из листе су дељене (подразумевано)
`private (var_list)` - све променљиве из листе су приватне

Уколико желимо да променљиве **x** и **y** буду недељене, иако су декларисане изнад линије у којој се налази директива, то ћемо урадити на следећи начин:

```
    #pragma omp parallel private(x,y)
```

Приватне промењиве су неиницијализоване на почетку паралелног региона. Ако желимо да омогућимо иницијализацију **x** и **y** са вредностима које су биле додељене пре директиве, можемо користити кључну реч **firstprivate**. Такође, ако желимо да ажурирамо промењиве користећи вредност из последње итерације, користићемо **lastprivate**.

У наставку следи пример коришћења **private** кључних речи.

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

int main(void){
    int i;
    int x;
    x = 44;
#pragma omp parallel for private(x)
    for(i = 0; i <= 10; i++){
        x = i;
        printf("Thread num: %d  x: %d\n",omp_get_thread_num(),x);
    }
    printf("x is %d\n", x);
}
```

Након покретања програма, добијамо следеће:

```
Thread num: 0     x: 0
Thread num: 0     x: 1
Thread num: 0     x: 2
Thread num: 3     x: 9
Thread num: 3     x: 10
Thread num: 2     x: 6
Thread num: 2     x: 7
Thread num: 2     x: 8
Thread num: 1     x: 3
Thread num: 1     x: 4
Thread num: 1     x: 5
x is 44
```

Приметићете да промењива **x** има исту вредност на коју је иницијализована пре коришћења паралелне регије. Претпоставимо да смо желели задржати последњу вредност промењиве **x** након паралелне регије. То се може постићи коришћењем **lastprivate**. Уколико замените **private(x)** са **lastprivate** (**x**) добићемо следећи резултат:

```
Thread number: 3     x: 9
Thread number: 3     x: 10
Thread number: 1     x: 3
Thread number: 1     x: 4
```

```
Thread number: 1      x: 5
Thread number: 0      x: 0
Thread number: 0      x: 1
Thread number: 0      x: 2
Thread number: 2      x: 6
Thread number: 2      x: 7
Thread number: 2      x: 8
x is 10
```

Добијена вредност за **x** је 10, а не 8. Зашто? То значи да је последња итерација задржана, а не последња операција. Шта ће се десити ако заменимо **lastprivate** (**x**) са **firstprivate** (**x**)?

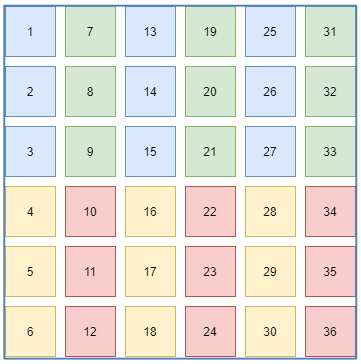
```
Thread number: 3      x: 9
Thread number: 3      x: 10
Thread number: 1      x: 3
Thread number: 1      x: 4
Thread number: 1      x: 5
Thread number: 0      x: 0
Thread number: 0      x: 1
Thread number: 0      x: 2
Thread number: 2      x: 6
Thread number: 2      x: 7
Thread number: 2      x: 8
x is 44
```

Многи би у овом случају очекивали вредност 0, тј. да променљива **x** има вредност као при првој итерацији. Коришћење **firstprivate** кључне речи подразумева да свака нит има сопствену инстанцу променљиве која би требала бити иницијализована на вредност променљиве, јер постоји пре паралелне конструкције. У том случају, свака нит добија своју инстанцу **x** и она је једнака 44.

Одредбе за контролу распоређивања:

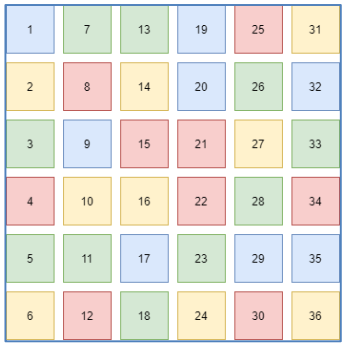
Како би се успешно оптимизовао код, битно је одредити како се итерације распоређују по нитима. Циљ је таква дистрибуција посла која за резултат има минимизовање времена извршавања. ОпенМП нуди три начина распоређивања код паралелних петљи:

1. schedule (*static*, величина_блока) – други параметар дефинише величину блокова на које су итерације подељене. Итерације се равномерно распоређују по нитима. Уколико је дефинисана величина блока, количина посла се дели на сегменте те величине. Уколико је на располагању H нити, свака нит добија H-ти део посла и блокови се статички додељују нитима. Распоређивање се врши редом у круг. Ово је подразумевана опција.



```
#pragma omp parallel for schedule (static, 3)
for(int i = 0; i < 36; i++)
{
    Work(i);
}
```

2. schedule (*dynamic*, величина_блока) – итерације су подељене у блокове чија је величина одређена другим параметром. Читава петља се дели на блокове и они се смештају у ред. Блокови се динамички распоређују на извршавање по нитима, односно чин једна нит нема посла, преузима први блок из реда. Подразумевана вредност другог параметра (уколико се изостави) је 1.




```
#pragma omp parallel for schedule (dynamic, 1)
for (int i = 0; i < 36; i++)
{
    Work(i);
}
```

3. schedule (*guided*, величина_блока) – дефинише се динамичко заказивање блокова са опадајућом величином. Иницијално је величина блока већа и смањује се све до величине дефинисане другим параметром. Погодно је за случајеве када су нити међусобно у временском раскораку, где свака итерација захтева отприлике исту количину посла за обраду.

Концепт секција:

У неким ситуацијама независни делови посла могу бити извршавани конкурентно. На пример, мењамо вредности два вектора независно један од другог. У том случају, нит ћемо задужити да обавља обе операције у паралели. Ово се обавља коришћењем секција. Компајлер заказује конкурентно извршавање кода унутар секције. Оваква конструкција омогућава доделу другачије обраде свакој нити.

У примеру који слединаправљен је паралелни регион. Унутар региона дефинисан је блок са секцијама користећи директиву:

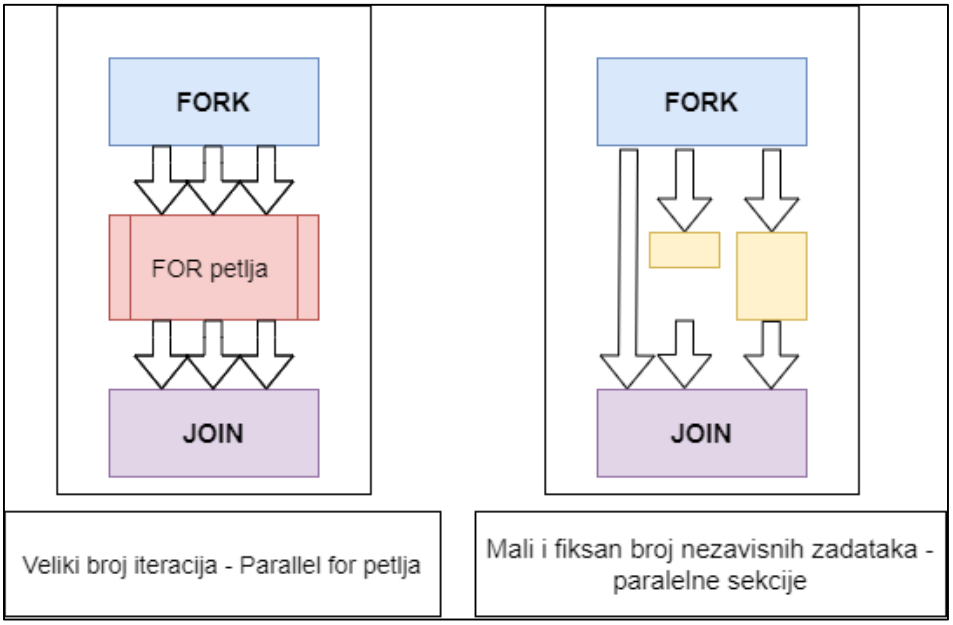
```
#pragma omp sections
```

Након тога свака од направљених секција представља део кода који се може извршавати у паралели са преосталим секцијама у блоку. Свака секција се распоређује на једну од нити из скупа нити за дати пралелни регион.

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

        #pragma omp section
        y_calculation();

        #pragma omp section
        z_calculation();
    }
}
```

За директиву за блок секција важе једнака правила за одредбе за дељење променљивих. Такође, као код паралелне петље, могуће је написати блок секција који је уједно и паралелни регион користећи директиву:

```
#pragma omp parallel sections
```



Слика 2 - Паралелне секције

Сихронизација нити

Као што је и приказано у претходним примерима, након сваког паралелног региона, паралелне петље или блока са паралелним секцијама чека се да све нити из скупа заврше посао пре него што се настави са даљим извршавањем кода. Ова тачка синхронизације где нити чекају на завршетак осталих назива се **баријера** (енгл. *barrier*).

Баријере је могуће поставити и експлицитно, унутар било ког паралелног региона, употребом директиве:

```
#pragma omp barrier
```

На следећем примеру приказано је коришћење баријере. Паралелни регион започиње тако што све нити из скупа извршавају позив функције *big_calc1* при чему прослеђују свој идентификатор као параметар функције. Потом се чека да све нити заврше обраду и попуне низ A, пре него што иједна нит изврши позив наредне функције. Да је у примеру изостављена баријера, свака нит би извршила позив функције *big_calc2* одмах по завршетку претходне.

```
#pragma omp parallel shared (A, B) private(id)
{
    id=omp_get_thread_num();
    A[id] = big_calc1(id);

    #pragma omp barrier

    B[id] = big_calc2(id, A);
}
```

Баријере које су имплицитно постављене на крај паралелног региона, петље или блока са секцијама, могуће је уклонити користећи одредбу ***nowait*** приликом прављења. Уколико ова одредба стоји у директиви, то значи да ће свака нит одмах по завршетку свог посла наставити са даљим извршавањем кода, без чекања на остале. Следи пример.

```
#pragma omp parallel shared (A, B, C) private(id)
{
    #pragma omp for nowait
    for(i = 0; i < N; i++)
    {
        A[i] = big_calc1(i);
    }

    #pragma omp for
    for(i = 0; i < N; i++)
    {
```

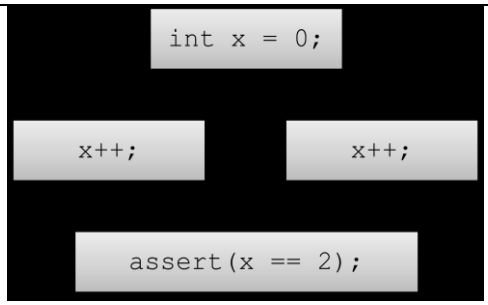
```
        B[i] = big_calc2(i);
    }

    #pragma omp for
    for(i = 0; i < N; i++)
    {
        C[i] = big_calc3(A, B, i);
    }
}
```

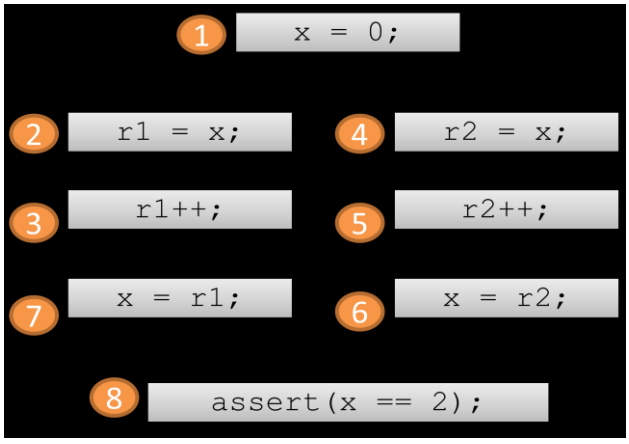
У примеру прва и друга петља су независне. Пошто уз прву директиву стоји одредба *nowait* то значи да ће одмах по завршетку свог посла, свака нит започети извршавање наредне петље. Трећа петља за рачунање користи резултат претходне две, те је неопходно сачекати да се претходне две заврше пре него што започне. Изостављањем одредбе *nowait* имплицитно се умеће баријера након друге петље.

Трка до података

До трке до података долази када две логички паралелне инструкције приступају истој меморијској локацији и бар један од та два приступа је ради писања. Тада је вредност те меморијске локације неодређена. Глобалне (дељене) промењиве су чест узрок трке до података. Пример:

```
int x = 0;
#pragma omp parallel for
for (int i = 0; i < 2; i++)
{
    x++;
}
assert(x == 2);
```





Слика 3- Приказ трке до података

На слици 3 дат је мало детљенији приказ онога шта се дешава. Бројеви означавају редослед инструкција којим се оне извршавају. Овај поредак није увек исти, јер кораци 2, 3 и 7 се извршавају независно од корака 4, 5 и 6. У овом примеру, регистар *r2* преузима вредност променљиве *x* пре него што је она записана (корак 7), те има вредност 0. У кораку 6 и у кораку 7 вредност која је уписана у *x* је број 1, што ће бити и коначан резултат након завршетка петље.

Приликом увођења параелизма у програмски код неопходно је водити рачуна да се овакве ситуације избегну. Код ОпенМП конструкција, потребно је водити рачуна да све итерације паралелне петље, као и све секције унутар једног блока са секцијама буду међусобно независне. За сваку дељену променљиву унутар било ког паралелног региона потребно је обезбедити да јој максимално једна нит приступа зарад писања.

У случају да је неопхоцно да више нити приступају једној променљивој, овај део кода назива се критична секција. ОпенМП нуди неколико начина који обезбеђују да максимално једна нит приступа критичној секцији истовремено. Први начин јесте **директива за критичну секцију**:

```
#pragma omp critical (name)
```

Ова директива мора се наћи унутар паралелног региона. Односи се на наредни блок кода (или инструкцију), и означава да у једном тренутку само једна нит може да извршава дати блок кода. Уколико нека друга нит из скупа дође до тог блока, чекаће да претходна нит заврши извршавање секције. Директиви се као параметар може проследити назив секције. У том случају, у једном тренутку, само једна нит може да извршава било коју од секција са датим именом.

На следеће мпримеру дато је како би изгледала заштита критичне секције из претходног примера:

```
int x = 0;
#pragma omp parallel for
for (int i = 0; i < 2; i++)
{
    #pragma omp critical
    x++;
}
assert(x == 2);
```

Следећи пример приказује заштиту више критичних секција које су међусобно независне, користећи именовање критичних секција. У примеру, паралелни регион као

дељене промењиве прима редове *x* и *y*. Свака нит преузима одговарајући задатак из једног реда, изврши га, па потом преузима задатак из другог реда и изврши и њега. Користећи директиву за критичну секцију обезбеђено је да само једна нит може да ради преузимање задатка из једног реда како се не би десило да две нити преузму исти задатак. Именовањем секција обезбеђено је да док једна нит преузима задатак из реда *x*, нека друга нит сме да преузима задатак из реда *y* (две незавнисе критичне секције).

```
#pragma omp parallel shared(x, y) private(x_next, y_next)
{
    #pragma omp critical (section_x)
     x_next = pop(x);

     work(x_next);

    #pragma omp critical (section_y)
    y_next = pop(y);

    work(y_next);}
}
```

У примеру на страници 14, дат је приказ како уз помоћ директиве за критичну секцију можемо решити проблем када више нити истовремено додаје вредности на дељену промењиву. Међутим ово решње уноси чекање на приступ критичној секцији, кад год се нека друга нит налази у њој. Уколико, као у примеру, критична секција представља читаво тело петље, то ће довести до неефикасног кода. Време извршавања ће бити приближно једнако секвенцијалном коду (у неким случајевима и спорије). Да би се овај проблем решио, ОпенМП нуди још један концепт који се назива редукујући хиперобјекти или **редуктори**.

Редуктори се могу дефинисати за оне операције које су асоцијативне. Унутар паралелног региона, уместо да све нити приступају једној дељеној промењивој, свака од нити заправо добије копију промењиве (као да је у питању приватна промењива), која је иницијализована на неутрални елемент за дату операцију (0 за сабирање, 1 за множење итд.). Свака нит врши обраду над својом локалном копијом. Током трајања извршавања датог паралелног региона, петље или блока са секцијама, вредност дељене промењиве је недефинисана. У тренутку када се стигне до краја, врши се ажурирање вредности дељене промењиве редуковањем привремених копија. Редуковање заправо прдставља примену исте операције над привремеим резултатима. У случају да је за дати регион, петљу или блок са секцијама задата одредба *nowait* редукција се врши на првој наредној баријери.

Редуктори се дефинишу на почетку паралелне конструкције додавањем одговарајуће одредбе. Одредба као параметар прима тип операције која ће се вршити над промењивом и листу промењивих.

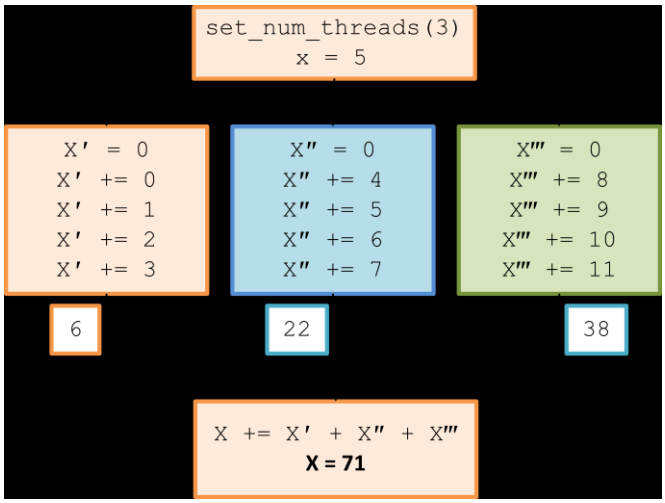
```
#pragma omp parallel reduction(op:variable-list)
```

Операције које се могу проследити су као параметар *op*: +, *, -, &, ^, |, &&, ||. Над редуктором једног типа могуће је вршити само операцију тог типа.

Пример:

```
omp_set_num_threads(3);

int x = 5;
#pragma omp parallel for schedule (static, 4) reduction(+, x)
for (int i = 0; i < 12; i++)
{
    x += i;
}
cout << x;
```

У примеру је направљен скуп од 3 нити, потом је направљена паралелна петља која користи статичко распоређивање са величином бока 4. Променљива *x* је прослеђена као редуктор за сабирање. То значи да ће бити се петља поделити на 3 блока дужине по 4 итерације, и сваки од блокова ће се извршити на једној од нити. Све 3 нити добиће привремену променљиву *x* иницијализовану на 0.



Слика 4 - Приказ рада редуктора

Свака од нити врши обраду над својом локалном копијом. Након завршетка паралелне петље (на имплицитно уметнутој баријери) врши се редуковање резултата тако што се све привремене копије саберу, и додају на променљиву *x*.

Задаци:

У прилогу се налази 6 примера. У сваком примеру је дат секвенцијални код, и исти код паралелизован употребом ОпенМП директива. Потребно је дате примере отворити, превести и покренути. За сваки од примера потребно је одговорити на следећа питања која следе.

Пре почетка направите нову датотеку са називом ***ImePrezimeRA###_Vežba4.txt***. На почетку датотеке написати име и презиме, након тога у новом реду окружење које сте користили за превођење, оперативни систем који користите, модел процесора на коме се решење извршава и број језгара. Након тога, редом пишите одговоре на питања.

За сваки пример где се тражи мерење времена (убрзања) водити рачуна да користите конфигурацију ***Release*** приликом превођења.

Пример 1: пример паралелног региона и добављања идентификационог броја нити

1. Шта је испис након извршења секвенцијалног кода?
2. Шта је испис након извршења паралелног кода?
3. Колико нити је парављено у паралелном примеру?
4. Напишите измењену функцију *main* тако да направи дупло мање нити од максималног броја дозвољених.

Пример 2: пример паралелне петље. Пример одговара примеру датом на страни 6.

1. Које је време извршавања секвенцијалног кода?
2. Које је време извршавања паралелног кода?
3. Колико је постигнуто убрзање?
4. Да ли је убрзање једнако броју направљених нити? Зашто?

Пример 3: пример коришћења наредбе *nowait.*

1. Покренути секвенцијални код. Покренути паралелни код више пута. Да ли су излаз из секвенцијалног и паралелног кода исти? Зашто?
2. Шта је потребно урадити да би се грешка исправила?
3. Колико је било убрзање пре исправке? Колико је убрзање након исправке?
4. Шта можете урадити да поправите убрзање након исправке?

Пример 4: пример коришћења редуктора.

1. Да ли је излаз и серијског и паралелног кода исти?
2. Обрисати одредбу *reduction(..)* на линији 22 у паралелној верзији кода. Да ли је резултат исти? Зашто?

3. Изменити код тако да заштитите упис у променљиву користећи директиву за каритичну секцију (уместо редуктора). Као одговор на питање (након измена) исписати део кода између позива функција за добављање почетног и крајњег времена извршавања.
4. Колико је убрзање кода када се користи редуктор, а колико када се користи критична секција?

Пример 5: пример коришћења директиве за критичне секције. Илуструје да приступ променљивој није једини узрок критичне секције.Овде проблем настаје због конкурентног уписа у стандардни излаз.

1. Покрените паралелни код Више пута, и погледајте излаз. Шта је проблем?
2. Измените код користећи директиве за критичне секције тако да обезбедите да у сваком реду буде исписана само једна реченица. Али тако да и даље нити исписују реченице произвољним редоследом (само је тело петље критична секција, не читава петља). Као одговор на питање напишите код функција *print1*, *print2* и *print3* након измена.

Пример 6: пример рачунања n-тог броја Фибоначијевог низа користећи рекурзивни алгоритам. Пример илуструје како је могуће раздвојити посао тако да га извршавају две нити у паралели.

1. Покрените серијски и паралелни код. Да ли је резултат исти?
2. Упоредите брзине извршавања, колико убрзање је постигнуто применом паралелизма?
3. Смањите *n* на 20. Колико је сада убрзање? Зашто?

Додатни задатак (напредни):

1. Изменити пример 6 тако да се посао раздвоји на онолико нити колико је максимално доступно на систему.
   Помоћ: можете користити дељену променљиву која представља број доступних нити. У пратећој документацији за ОпенМП проверите значење функције *omp_set_nested*