Паралелно програмиране с OpenMP

Цел:

- Запознаване с примерни програми, използващи паралелно с програмиране с многонишков програмен модел, компилиране, настройване и изпълнение на OpenMP програми.

Очакван резултат:

Писмено представяне на експерименталните резултати и анализи във вид на електронен протокол.

Теоретична част:

- **Приложен програмен интерфейс OpenMP**

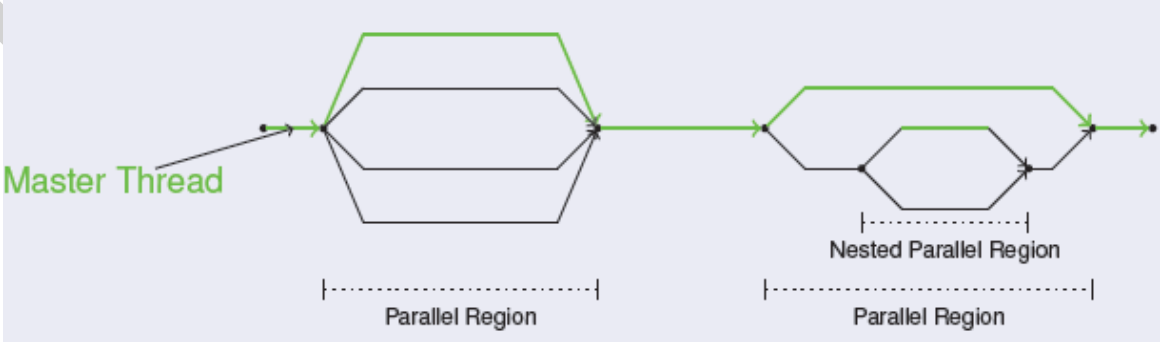
OpenMP е съвкупност от широко използвани стандарти и интерфейси за разработка на паралелни приложения за системи с общодостъпна памет. Поддържа се от множество производители на компилатори: GNU, Intel, IBM, Oracle, PGI, Sun, Cray, Fujitsu, HP, и др. Вж.

Създаването на паралелни приложения с помощта на технологията OpenMP предполага използването на базови езици като C/C++ и FORTRAN. Изходната програма се създава на основата на избрания базов език за програмиране, като се добавят допълнителни директиви на компилатора за OpenMP, наричани ***прагми*** (**pragmas**).

- **Модел на изпълнение**

Моделът на изпълнение на една OpenMP програма се основава на пораждането и обединението на нишки (fork-join). Основната нишка (master thread) създава група нишки, които се събират в края на даден паралелен участък (Фиг. 1.1). В дадена паралелна програма може да има произволно количество паралелни области. Те от своя страна могат да бъдат влагани една в друга.

Такъв модел на изпълнение може да бъде приспособен за решаване на редица задачи, но неговата реализация в OpenMP е насочена към разпаралеляването на програми, в които основната работа се извършва в *цикли*, в частност програми, обработващи големи масиви от данни.



Фиг. 1.1: Паралелизъм “FORK-JOIN”.

- **Модел на паметта**

Технологията OpenMP предполага, че програмата се изпълнява в система с общодостъпна памет, в която могат да бъдат съхранявани избрани променливи от програмата, достъпни за всички нишки. Всяка нишка има достъп до памет, която не е достъпна за останалите нишки, така наречената частна (private) памет за нишката.

Променливите в едно OpenMP приложение могат да бъдат *споделени* (shared) и *локални* (private). Споделените променливи са видими за всички нишки. Ако една нишка промени една споделена променлива, то всички останали нишки ще регистрират промяната. Всички променливи, създадени преди създаването на самите нишки, са споделени.

Локалните променливи са видими са само за нишката, която ги притежава. Промяна, направена в private променлива, се вижда само от нишката, която притежава тази променлива. Локално създадените променливи са винаги *private*.

- **Синтаксис на OpenMP директиви**

Спецификацията OpenMP осигурява широк спектър от директиви на компилатора, т. нар. **прагми**, необходими за паралелизацията и организацията на програмата. Те представляват директиви на компилатора, чиято грижа е създаването на нишки и управлението на паметта. Всяка една прагма поддържа различни клаузи за: *управление на паметта*; *разпределение на нишки*; *управление на нишки*; *логически проверки* и *синхронизация*.

#pragma omp <тип> [клауза[[,] клауза] ...]

В OpenMP се поддържат два основни вида паралелизъм:

- *на ниво цикли* (loops) - всяка нишка получава уникален обхват от стойностите на индексираната променлива.
- *на ниво секции* (sections) - паралелизират се участъци, които могат да бъдат произволни редове от програмния код.

OpenMP позволява създаване на произволен брой нишки (максималният брой зависи от платформата), но обикновено оптималният брой е равен на броя на физически процесори (ядра), който може да се получи чрез функция **omp_get_num_procs().**

Директиви за задаване на паралелна област

- **Директива PARALLEL** - определя паралелен участък, който да се изпълни от няколко нишки. Компилаторът създава съответните нишки.

```
#pragma omp parallel [clause[[,]clause] ...] new-line
   structured-block
```

Директиви за разпределяне на работата: *разпределение на ниво цикли*

- **Разпределение на ниво цикли**

  Директива FOR - указва на компилатора да изпълни итерациите на даден for цикъл паралелно.

```
#pragma omp for [clause[[,] clause] ... ] new-line
   for-loop
```

- **Разпределение на ниво секции**

  Директива SECTIONS -

```
#pragma omp sections [clause[[,] clause]...] new-line
{
```

```
    [#pragma omp section new-line]
      structured-block
    [#pragma omp section new-line
      structured-block ]
...
}
```

- **Директива SINGLE** - определя секция от кода, която трябва да се изпълни само от една нишка.

```
#pragma omp single [clause[[,] clause]...] new-line
structured-block
```

Комбинирани директиви

```
#pragma omp parallel for [clause[[,] clause]...] new-line
for-loop
```

```
#pragma omp parallel sections [clause[[,]clause]...]new-line
{
    [#pragma omp section new-line]
      structured-block
    [#pragma omp section new-line
      structured-block ]
    ...
}
```

Директиви за синхронизация

- **Директива MASTER** - синхронизираща прагма, която указва, че само главната нишка трябва да изпълни този код **.**

```
#pragma omp master new-line
structured-block
```

- **Директива CRITICAL** - определя част от кода, която трябва да се изпълни само от една нишка в даден момент от време.

```
#pragma omp critical [(name)] new-line
structured-block
```

- **Директива BARRIER** – синхронизираща прагма, която указва, че всички нишки трябва да достигнат до тази точка в кода, за да продължи неговото изпълнение нататък.

```
#pragma omp barrier new-line
```

- **Директива ATOMIC** – определя област от паметта, която може да се промени само от една нишка в даден момент.

```
#pragma omp atomic new-line
```

- **Основни OpenMP клаузи**

Клаузите са опционални модификатори на действието на директивите.

Клауза	Описание
if (израз)	Паралелният регион ще се изпълни само ако израза в скобите е верен, в противен случай се изпълнява последователно.
private (променлива1, …)	Декларира, че променливите в скобите ще бъдат private за всяка нишка.
firstprivate (променлива1, …)	Начална стойност на private променлива от списъка на клаузата за всяка нишка е стойността, която променливата има за нишката непосредствено преди достигане на паралелната конструкция.
lastprivate (променлива1, …)	Стойността на променлива от списъка на клаузата след приключване на паралелната конструкция е стойността, получена на последната последователна итерация на цикъла или последната лексическа директива section.
num_threads (int)	Указва колко нишки да изпълнят паралелния регион
shared (променлива1, …)	Декларира кои променливи ще са общи за всички нишки
default (shared | none)	Декларира, какви да са променливите създавани при изпълнението на нишката
shared	Всички създадени променливи ще са общи
none	Всички създадени променливи ще са private.
reduction ({+|-|*|/} : променлива1, …)	Извършва редукция върху изброените скаларни променливи. Условията за променливите са: (a) Трябва да са от подходящ тип.(b) Трябва да са общи за всички нишки. (c) Не трябва да са указатели
nowait	Използва се заедно с директива for, премахва бариерната синхронизация на нишките в края на цикъла.
schedule (тип)	Определя как да бъдат разпределяни между нишките итерациите на цикъла for:
dynamic	Итерациите се разпределят между нишките на принципа First Come First Server,
static	Итерациите се разделят на равни части и всяка нишка обработва своята част.
guided	Итерациите се разделят на части с различна големина и се обработват от нишките на принципа First Come First Server.
runtime	Задава се по време на изпълнението на програма с променливата OMP_SCHEDULE.

- **Основни функции**

Име	Описание
void omp_set_num_threads(int num_threads)	Задава с колко нишки да се изпълняват паралелните региони.
int omp_get_num_procs()	Връща колко процесора (ядра) има системата.
int omp_in_parallel()	Връща резултат различен от нула, ако се извика от паралелен регион.
int omp_get_thread_num()	Връща номера на нишката, която изпълнява паралелния регион.
int omp_get_num_threads()	Връща броя на нишките, изпълняващи паралелния регион.
double omp_get_wtime()	Връща времето в секунди от даден постоянен минал момент.

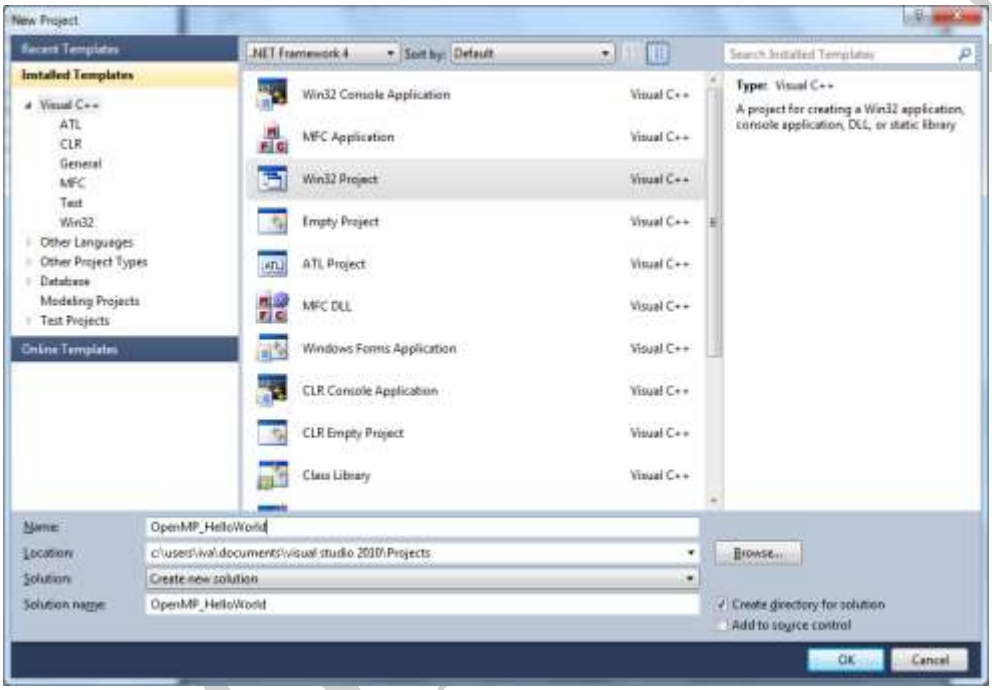
- **Променливи на средата**

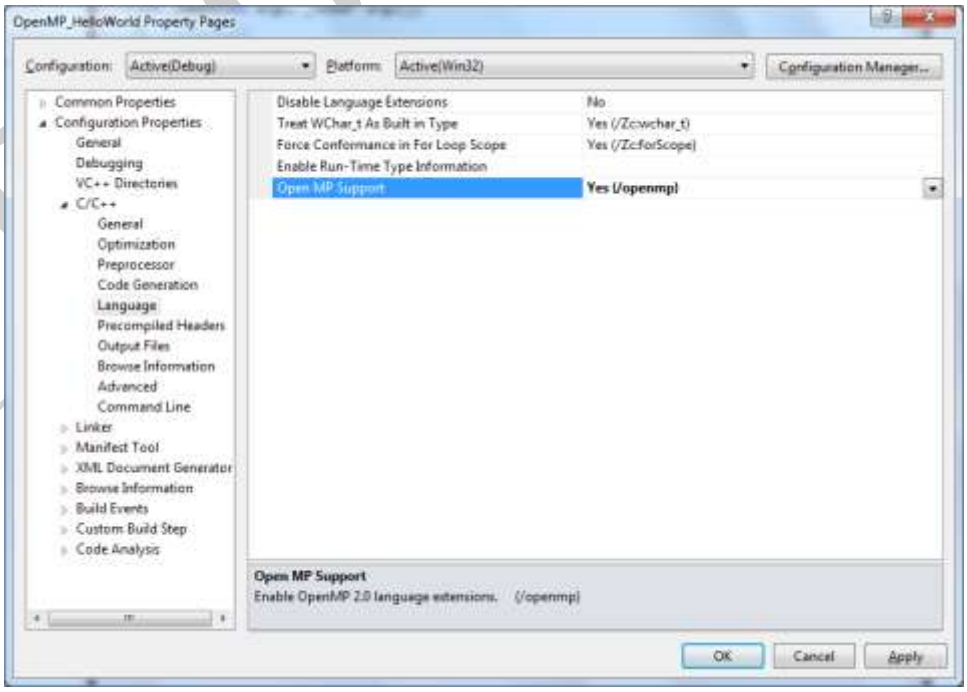
В OpenMP са предоставени няколко runtime променливи на средата, които могат да бъдат използвани за управление поведението на дадена *OpenMP*-програма. Най-важната и широко използвана от тях е **OMP_NUM_THREADS**, която определя броя на нишките, които ще бъдат създадени, когато основната нишка достигне до паралелен участък. Общото правило е този брой нишки да бъде равен на броя процесори в системата.

Изпълнение на експерименталната част:

A. Създаване, компилиране и изпълнение на OpenMP приложения

- **OpenMP приложение под Windows**





A.1. Въведете следния програмен код за конзолното приложение. Съхранете го под име **OpenMP_HelloWorld.cpp.**

```
// OpenMP_HelloWorld.cpp : Defines the entry point for the console application.
#include "stdafx.h"
#include <omp.h>
int _tmain(int argc, _TCHAR* argv[])
{
   int tid, nthreads, nprocs;
   // Sets the number of processors that are available to the program
   nprocs = omp_get_num_procs();
   printf("Number of processors = %d\n", nprocs);

   //Sets the number of threads in subsequent parallel regions to be equal to the
number of logical processors on the machine. For example, if you have a machine with
one physical processor that has hyperthreading enabled, it will have two logical
processors and, therefore, two threads.
      omp_set_num_threads(nprocs);
   //
   #pragma omp parallel private(nthreads, tid)
   {
         // Determine the number of threads that will be deployed for the parallel
region
         tid = omp_get_thread_num();
         printf("\nHello World is being printed by the thread id %d\n", tid);

         // The master thread of the team is thread 0
         if (tid == 0)
         {
               printf("\nMaster prints Num of threads \n");
               nthreads = omp_get_num_threads();
               printf("Total number of threads = %d\n", nthreads);
         }
   }
   return 0;
}
```

A.2. Компилирайте програмата и я изпълнете. Анализирайте получения резултат.

Документирайте изпълнението на задачата и получения резултат в отчета си.

- **OpenMP приложение под Linux**

Изчислителният ресурс, използван за провеждане на лабораторните експерименти е хетерогенен компютърен клъстер. Той се състои от свързани в мрежа многоядрени сървъри, които могат да работят съвместно по обща задача. Потребителят се свързва към входен възел, през който се подават задачи за изпълнение на изчислителните възли (Фигура 1.2).

A.3. Преработете дадения в A.1 код като замените редове:

```
#include "stdafx.h"
#include <omp.h>
int _tmain(int argc, _TCHAR* argv[])
```

с

```
#include <stdio.h>
#include <omp.h>
int main(int argc, char* argv[])
```

A.4. Съхранете направените промени в нов файл под името **OpenMP_HelloWorld_cluster.cpp.**

A.5. Осъществете достъп с клъстера с помощта на програма за отдалечен достъп. Въведете потребителско име и парола.

A.6. В работната директория на входния възел на клъстера, създайте директория „**PP_OpenMP**“. С помощта на програмата за файлов трансфер, прехвърлете файла **OpenMP_HelloWorld_cluster.cpp**.

```
mkdir PP-OpenMP
cd PP-OpenMP
```

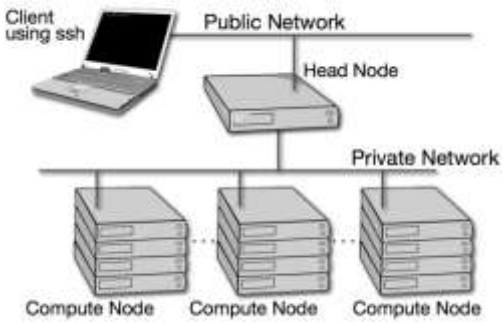
A.7.Компилирайте изходния програмен код

```
g++ -fopenmp -o OpenMP_HelloWorld OpenMP_HelloWorld_cluster.cpp
```

A.8.Изпълнете програмата и анализирайте получения резултат.

```
./OpenMP_HelloWorld
```

Документирайте изпълнението на задачата и получения резултат в отчета си.



Фиг. 1.2: Достъп до компютърен клъстер по ssh протокол .

Извличане на информация за изчислителната среда

A.9. Модифицирайте изходния код на програма **OpenMP_HelloWorld**, така, че главната нишка да изведе следната допълнителна информация за изчислителната среда, в която се изпълнява програмата:

- Брой налични процесори;
- Брой използвани нишки;
- Максимален брой налични нишки;
- Извършва ли част от изчислителната работа в паралелен участък на програмата?
- Възможен ли е контрол на броя налични нишки в паралелен участък в процеса на изпълнение?
- Поддържали се вложеност на паралелни участъци?

Използвайте за целта следните библиотечни функции.

• `omp_get_num_procs()` • `omp_get_num_threads()` • `omp_get_max_threads()`	• `omp_in_parallel()` • `omp_get_dynamic()` • `omp_get_nested()`

A.10. Съхранете модифицираната програма под име "**omp_getEnvInfo.cpp**".

```
// omp_getEnvInfo.cpp: Defines the entry point for the console application.

#include "stdafx.h"
#include <omp.h>


int _tmain(int argc, _TCHAR* argv[])
{
      int nthreads, tid, procs, maxt, inpar, dynamic, nested;

// Start parallel region /
#pragma omp parallel private(nthreads, tid)
```

```
  {

  // Obtain thread number
  tid = omp_get_thread_num();

  // Only master thread does this
  if (tid == 0)
    {
    printf("Thread %d getting environment info...\n", tid);

    // Get environment information
    procs = omp_get_num_procs();
    nthreads = omp_get_num_threads();
    maxt = omp_get_max_threads();
    inpar = omp_in_parallel();
    dynamic = omp_get_dynamic();
    nested = omp_get_nested();


    // Print environment information
    printf("Number of processors = %d\n", procs);
    printf("Number of threads = %d\n", nthreads);
    printf("Max threads = %d\n", maxt);
    printf("In parallel? = %d\n", inpar);
    printf("Dynamic threads enabled? = %d\n", dynamic);
    printf("Nested parallelism supported? = %d\n", nested);
    }

  }  // Done
      return 0;
}
```

A.11. Компилирайте и изпълнете програмата, първо на платформата по Windows, а след това на изчислителния клъстер под Linux. Анализирайте получените резултати.

A.12. Повторете експерименталните изследвания на примерния програмен код с вариране на броя на генерираните нишки и анализирайте отново получените резултати.

Документирайте изпълнението на задачата и получения резултат в отчета си.

B. Разпределяне на обработката между множеството нишки: *паралелизъм на ниво цикли*

Създайте програма, която използва директива OpenMP FOR за задаване паралелно изпълнение на итерациите на цикъл for при задаване на различни стратегии - *dynamic*, *guided*, *runtime* and *static* за разпределяне на итерациите между паралелно работещите нишки. Всяка нишка изпълнява възложената и съвкупност от итерации (CHUNKSIZE) преди да и бъде възложена следваща.

B1. Изследвайте експериментално показания по-долу код като анализирате по какъв начин даден се използват OpenMP директиви и библиотеки за осъществяване на динамично планиране, разпределението на итерациите на цикъл между паралелно работещи нишки. Въведете кода и го съхранете във файл „**omp_workshare.cpp**“.

```
// File: ”omp_workshare.cpp”
#include <omp.h>
#include <stdio.h>
```

```
#define CHUNKSIZE   10
#define N       100

int main (int argc, char *argv[])
{
      int nthreads, tid, i, chunk;
      float a[N], b[N], c[N];


      // Some initializations /
      for (i=0; i < N; i++)
            a[i] = b[i] = i * 1.0;
            chunk = CHUNKSIZE;

      #pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
      {
            tid = omp_get_thread_num();
            if (tid == 0)
            {
                  nthreads = omp_get_num_threads();
                  printf("Number of threads = %d\n", nthreads);
            }
            printf("Thread %d starting...\n",tid);

      #pragma omp for schedule(dynamic,chunk)
      for (i=0; i<N; i++)
         {
            c[i] = a[i] + b[i];
            printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
         }

      }  // end of parallel section
      return 0;
}
```

B2. Осъществете достъп с клъстера с помощта на програма за отдалечен достъп. Въведете потребителско име и парола.

B3. С помощта на програмата за файлов трансфер, прехвърлете файла **omp_workshare.cpp** в поддиректория „**PP_OpenMP**“ на основната директория на входния възел.

B4. Компилирайте и анализирайте получения от изпълнението на кода резултат.

- Как се поделят итерациите на цикъла между нишките при динамична стратегия за планиране на нишките?

```
$./omp_schedule | sort
```

B5. Повторете изпълнението на програмата още няколко пъти и анализирайте и обобщете получаваните резултати.

Документирайте изпълнението на задачата и получените резултати в отчета си.

B6. Модифицирайте кода на програмата, като подмените клауза **dynamic** на директива **schedule** с клауза **static**.

B7. Компилирайте кода на програмата отново и изпълнете променената програма.

- Съпоставете резултата с този, получен при изпълнението на точка B4.

B8. Повторете изпълнението на програмата още няколко пъти и анализирайте и обобщете получаваните резултати.

- Как се поделят итерациите на цикъла между нишките при статична стратегия за планиране на нишките?

Документирайте изпълнението на задачата и получения резултат в отчета си

B9. Повторете изпълнението на програмата още няколко пъти и анализирайте и обобщете получаваните резултати за стратегии *'guided'* и *'runtime'*. Прекомпилирайте и изпълнете модифицираната програма. Анализирайте получените резултати като ги съпоставите с тези от предишните изпълнения.

Документирайте изпълнението на задачата и получения резултат в отчета си

B. Разпределяне на обработката между множеството нишки: *паралелизъм на ниво участъци*

Създайте програма, която използва директива OpenMP SECTIONS за задаване на различни операции над масив от страна на всяка от нишките, изпълняващи се в паралелната секция.

B10. Изследвайте експериментално показания по-долу код като анализирате по какъв начин даден паралелен участък в програмата се разделя на отделни части, всеки от които ще бъде изпълнен от нишка. Създайте кода и го съхранете във файл „**omp_workshare2.cpp**".

```
// File: "omp_workshare2.cpp"

#include <omp.h>
#include <stdio.h>
#define N       50

int main (int argc, char *argv[])
{
      int i, nthreads, tid;
      float a[N], b[N], c[N], d[N];

      // Some initializations
      for (i=0; i<N; i++) {
            a[i] = i * 1.5;
            b[i] = i + 22.35;
            c[i] = d[i] = 0.0;
      }

      #pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
      {
            tid = omp_get_thread_num();
            if (tid == 0)
            {
                  nthreads = omp_get_num_threads();
                  printf("Number of threads = %d\n", nthreads);
            }
            printf("Thread %d starting...\n",tid);

            #pragma omp sections nowait
            {
                  #pragma omp section
                  {
                        printf("Thread %d doing section 1\n",tid);
                        for (i=0; i<N; i++)     {
                              c[i] = a[i] + b[i];
                              printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
                        }
                  }
```

```
                #pragma omp section
                {
                      printf("Thread %d doing section 2\n",tid);
                      for (i=0; i<N; i++)     {
                            d[i] = a[i] * b[i];
                            printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
                      }
                }
            } // end of sections
      printf("Thread %d done.\n",tid);
      }  // end of parallel section
      return 0;
}
```

B11. Осъществете достъп с клъстера с помощта на програма за отдалечен достъп. Въведете потребителско име и парола.

B12. С помощта на програмата за файлов трансфер, прехвърлете файла **omp_workshare2.cpp** в поддиректория „**PP_OpenMP**“ на основната директория на входния възел.

B13. Компилирайте и анализирайте получения от изпълнението на кода резултат.

- Колко нишки работят в паралелните секции?

B14. Повторете изпълнението на програмата още няколко пъти, анализирайте резултатите след всяко от изпълненията и обобщете получаваните резултати

Документирайте изпълнението на задачата и получения резултат в отчета си.

C. Задачи за самостоятелна работа

C1. Да се разработи програма на OpenMP за сумиране на елементите на едномерен масив с реални стойности. Наименувайте я ***„omp_sum_array.c“.*** Програмата трябва да получава като входни данни размерността на масива и извършва сумиране на елементите по два начина: *последователен* и *паралелен*.

- Елементите на масива да се генерират автоматично след подаване на желаната размерност при стартиране на програмата.
- За динамично резервиране на памет за масива използвайте функция **malloc**.
- За паралелното сумиране на всички елементи на масива приложете OpenMP директива ***parallel for*** и клаузите ***private*** и ***shared***, за определяне обхвата на променливите, използвани с паралелния участък.
- Осъществете достъп с клъстера и прехвърлете изходния програмен код файла в работна директория “**parprog**” на входния възел.
- Компилирайте и изпълнете примерния програмен код за последователно и паралелно сумиране елементите на масива с вариране на броя нишки.
- Анализирайте резултатите от изпълнението като сравните резултатите от последователното и паралелните изчисления. Обяснете причината при констатиране на несъответствие между резултатите от последователното и паралелното изчисление.
- Предложете и имплементирайте възможни решения на проблема. Проведете експерименти за верифициране на предложените решения.
- Модифицирайте кода на програмата като добавите таймери за измерване на времето за последователно и паралелно сумиране. За измерване на времето за изпълнение използвайте функция ***omp_get_wtime()***.

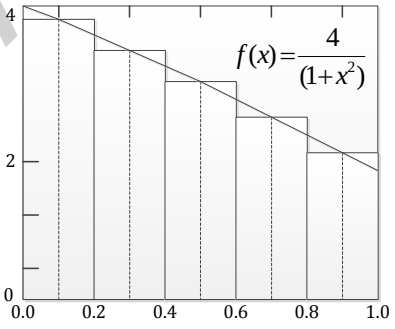
- Проведете експериментални изследвания за оценка на времето за изпълнение с вариране на броя нишки. Получените времеви резултати да се интерпретират в графичен вид като функция от броя нишки (*време_за_изпълнение = f (брой_нишки)*.

Документирайте изпълнението на задача C1 и получения резултат в отчета си.

C2. Да се разработи паралелна версия на последователната имплементация на програма, изчисляваща числото PI (π) чрез числено интегриране на функцията $f(x) = 4/(1 + x * x)$ в интервала от 0 до 1. Програмата трябва да може да приема като вход броят подинтервали за разделяне на интервала на интегриране, а като резултат да визуализира числената стойност на числото π. Да се проведат експериментални изследвания за съпоставяне времената за последователно и паралелно изпълнение. Да се проведат експерименти с вариране на броя нишки. Получените времеви резултати да се интерпретират в графичен вид като функция от броя нишки (*време_за_изпълнение = f (брой_нишки)*).

- За паралелизиране на алгоритъма за изчисление се използва разпределение на обработваните данни между нишките. Интервала на интегриране [0,1] се разпределя между нишките, така че всяка нишка определя локалния интеграл за даден под-интервал. Локалните изчисления на отделните нишки се обединяват, за да се получи крайния резултат. Числеността интегриране се осъществява с разделяне на интервала от 0 до 1 на n подинтервала и сумиране на площите на правоъгълниците под кривата на функцията на π (фиг. 1.4 – n = 5). По-големите стойности на n водят до получаване на по-точна апроксимация на стойността на π.

$$\int_0^1 \frac{1}{(1+x^2)} dx = \arctan(x)\Big|_0^1$$
$$= \arctan(1) - \arctan(0)$$
$$= \arctan(1)$$
$$= \frac{\pi}{4}$$



Фиг. 1.4: Числено интегриране на функция за определяне стойността на числото π.

Документирайте изпълнението на задача C2 и получения резултат в отчета си.

C3. Да се разработи паралелна версия на последователната имплементация на програма, изчисляваща интеграл от функцията $f(x) = 50/pi/(2500x2 + 1)$ в граници от 0 до 10. Да се проведат експериментални изследвания за съпоставяне на времената за изчисление с 1, 2 и 4 нишки. Получените времеви резултати да се интерпретират в графичен вид като функция от броя нишки.

Документирайте *изпълнението на задача C3 и получения резултат в отчета си.*