**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

отчет

**по лабораторной работе №5**

**по дисциплине «Параллельные алгоритмы»**

Тема: OpenMP

|  |  |  |
| --- | --- | --- |
| Студент гр. 6304 |  | Ковынев М.В. |
| Преподаватель |  | Митяков А.В. |

Санкт-Петербург

2018

**Цель работы.**

Знакомство с OpenMP.

**Задание.**

Параллельное вычисление числа Пи



Входные данные для программы:

* n - Количество слагаемых

Измерить время выполнения для количества потоков (1, 20).

**Основные теоретические положения.**

OpenMP - Открытый стандарт для распараллеливания программ на языках C, C++, Fortran. Стандарт интерфейса для многопоточного программирования над общей памятью. Компиляторы: GCC, Intel C++ Compiler, Visual C++.

Преимущества:

* Легкость использования
* Кросс-платформенность для систем с общей памятью
* Сокрытие низкоуровневых операций
* Поддержка параллельной и последовательной версий программ

Основные компоненты:

* Переменные окружения
* Директивы компилятора
* Функции

Порядок разработки программы:

1. Написать и отладить последовательную программу

2. Дополнить программу директивами OpenMP

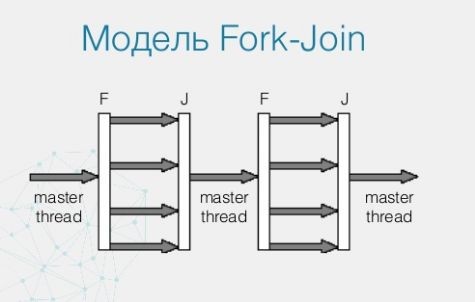
3. Скомпилировать программу компилятором с поддержкой OpenMP

4. Задать переменные окружения

5. Запустить программу

Модель работы:

* Явное указание параллельных секций
* Поддержка вложенного параллелизма
* Поддержка динамических потоков

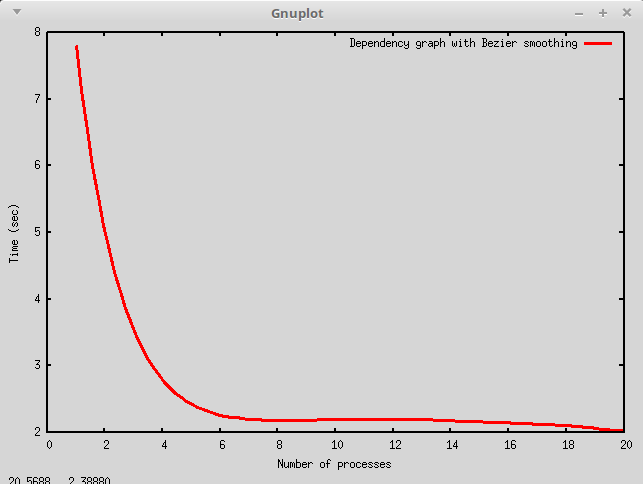


**Рисунок 1 — Модель работы OpenMP**

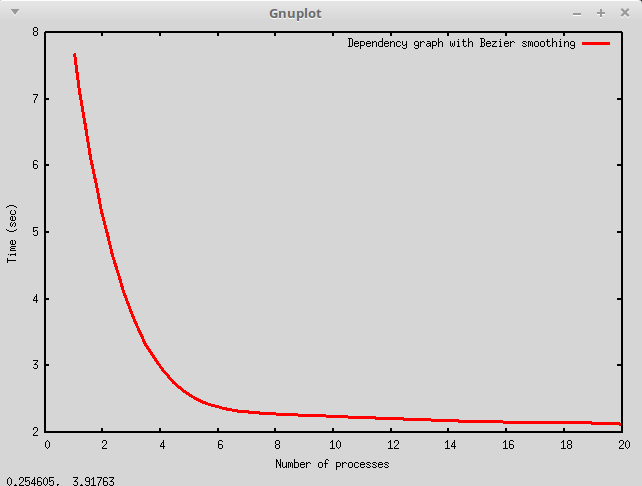
**Ход работы.**

В программе выделим параллельную секцию – цикл, который вычисляет сумму ряда. Обозначим секцию директивой *omp parallel*, а сам цикл директивой *omp for.* Таким образом компилятор разделит итерации цикла между потоками. Каждый поток будет суммировать свою часть ряда в локальную переменную *loc\_res*, которая была обозначена директивой *omp private(loc\_res)*. В конце работы поток записывает результат в переменную *res* используя директиву атомарной операции *omp atomic*.

В реализации с использованием параметра *reduction* добавим к директиве цикла предложение *reduction(+:res)*. В цикле результат будет аккумулироваться в переменную *res.* Компилятор создаст для каждого потока приватную копию *res*, а в конце работы потока атомарно сложит результат в глобальную переменную *res* аналогично реализации программы без использования параметра *reduction*.

Далее для измерения времени работы программы для данного количества процессов, а также для перебора различного количества потоков и анализа зависимости времени исполнения от количества используемых потоков составим программу **run.sh** и запустим её. В результате получим следующие графики:

**Рисунок 2 – Зависимость времени работы от количества потоков без Reduction**



**Рисунок 3– Зависимость времени работы от количества потоков c Reduction**

Как видно из графика с ростом количества потоков время исполнения уменьшается, однако при использовании более 4 потоков время исполнения программы практически не меняется, что можно объяснить наличием 4 логических процессоров в машине, на которой запускалась программа. Исходный код приведён в приложениях А, Б, В.

Наличие параметра *reduction* практически не влияет на время работы программы. Возможно это связано с объемом используемых данных. Однако код программы с использованием *reduction* выглядит более читабельным и понятным.

**Вывод**

OpenMP предоставляет высокоуровневую абстракцию над потоками ОС, что позволяет легче и быстрее писать параллельные программы. OpenMP хорошо подходит для реализации математических вычислений.

Приложение А

PI.C

#include "omp.h"

#include "stdio.h"

#include "stdlib.h"

int main(int argc, char const \*argv[]) {

int num\_sum = atoi(argv[1]);

long double res = 0.0;

long double loc\_res = 0.0;

int sign;

#pragma omp parallel firstprivate(loc\_res) private(sign)

{

#pragma omp for

for (int i = 1; i < num\_sum; i++) {

sign = (i + 1) % 2 ? -1 : 1;

loc\_res += (long double)(sign \* 4) / (long double)(2 \* i - 1);

}

#pragma omp atomic

res += loc\_res;

}

printf("%.62Lf\n", res);

return 0;

}

Приложение Б

pi\_reduction.c

#include "omp.h"

#include "stdio.h"

#include "stdlib.h"

int main(int argc, char const \*argv[]) {

int num\_sum = atoi(argv[1]);

long double res = 0.0;

int sign;

#pragma omp parallel private(sign)

{

#pragma omp for reduction(+:res)

for (int i = 1; i < num\_sum; i++) {

sign = (i + 1) % 2 ? -1 : 1;

res += (long double)(sign \* 4) / (long double)(2 \* i - 1);

}

}

printf("%.62Lf\n", res);

return 0;

}Приложение В

**run.sh**

#!/bin/bash

gcc $1 -fopenmp

echo -e "# Num proc\tTime" > plot.txt

for (( i=1; i<=20; i++))

do

echo -n "Calculate using ${i} proc: "

export OMP\_NUM\_THREADS=${i}

(time ./a.out 23279256000) &> file.txt

echo -ne "${i}\t" >> plot.txt

time=`head -3 file.txt | grep -o -P '(?<=m).\*(?=s)'`

echo "${time}"

echo "${time}" >> plot.txt

done

rm -f file.txt a.out

gnuplot -e "set ylabel \"Time (sec)\"; set xlabel \

\"Number of processes\"; plot \"plot.txt\" with lines \

smooth sbezier lw 3 t \"Dependency graph with Bezier smoothing\"; \

pause -1"