**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

отчет

**по лабораторной работе №3**

**по дисциплине «Параллельные алгоритмы»**

Тема: XSI IPC

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

Санкт-Петербург

2018

**Цель работы.**

Знакомство с XSI IPC (System V IPC) и с параллельными операциями с матрицами.

**Задание.**

Параллельное умножение матрицы на вектор. Реализовать два способа - разделение по строкам и по столбцам. Создается N процессов, каждый из которых работает над своей частью матрицы. Матрица, вектор и результирующий вектор хранятся в разделяемой памяти. Родительский процесс сигнализирует о завершении всех процессов. В случае необходимости защитить разделяемую память при помощи семафора.

**Основные теоретические положения.**

XSI IPC:

* Очереди сообщений
* Разделяемая память
* Семафоры

Классические средства межпроцессорного взаимодействия не позволяют организовать обмен между процессами, выполняющимися в различное время.

Идентификаторы и ключи:

* Каждому XSI IPC соответствует идентификатор в системе
* Идентификатор - внутреннее имя IPC
* В качестве внешнего идентификатора выступает специальный **ключ,** который используется в системных вызовах

**Ход работы.**

Для многих методов матричных вычислений характерным является повторение одних и тех же вычислительных действий для разных элементов матриц. Данный момент свидетельствует о наличии **параллелизма по данным** при выполнении матричных расчетов и, как результат, распараллеливание матричных операций сводится в большинстве случаев к разделению обрабатываемых матриц между процессорами используемой вычислительной системы.

Существуют два основных способа разбиения матриц по процессорам:

* Ленточное разбиение матрицы
* Блочное разбиение матрицы

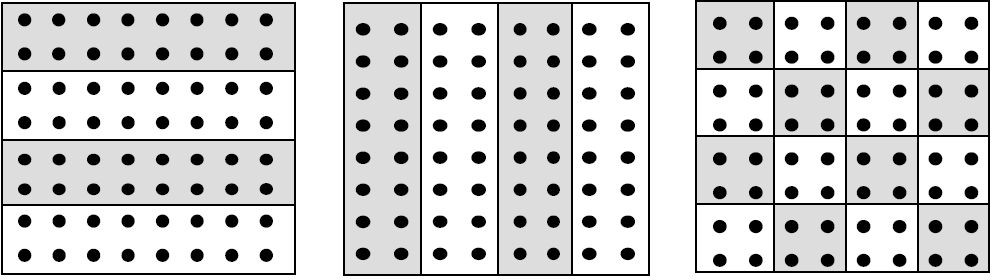
Способы распределения элементов матрицы:

Рисунок 1 – Способы разделения элементов матрицы

* Умножение матрицы на вектор при разделении данных по строкам:

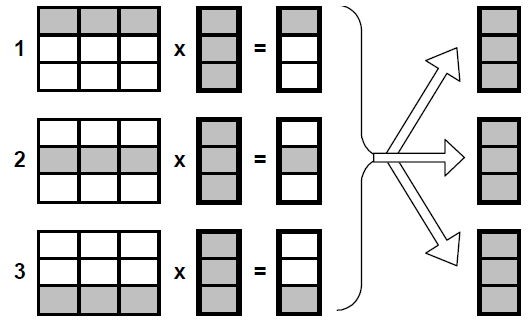
Для выполнения базовой подзадачи скалярного произведения процессор должен содержать соответствующую строку матрицы А и копию вектора b. После завершения вычислений каждая базовая подзадача определяет один из элементов вектора результата c.

Рисунок 2 – Разделение по строкам

* Умножение матрицы на вектор при разделении данных по столбцам:

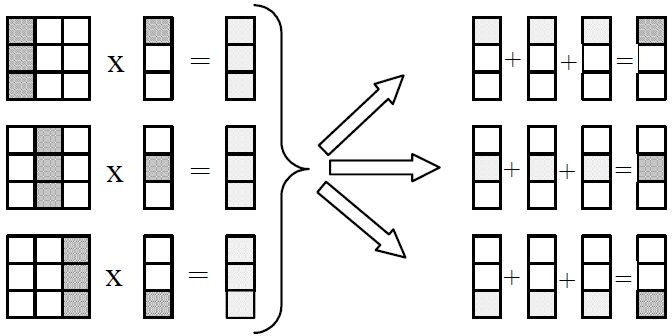
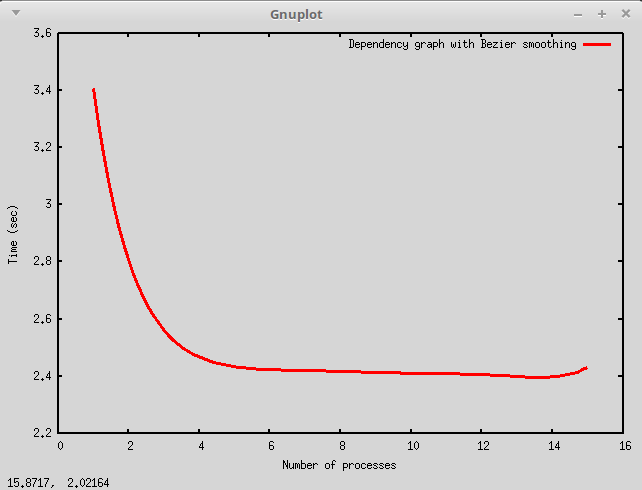
При таком способе разделения данных в качестве базовой подзадачи может быть выбрана операция умножения столбца матрицы **А** на один из элементов вектора **b**. Для организации вычислений в этом случае каждая базовая подзадача i, 0≤ i< n, должна содержать i-й столбец матрицы **А** и i-е элементы **bi** и **ci** векторов **b** и **с**.

Рисунок 3 – Разделение по столбцам

Реализуем две версии программы – соответственно при разделении по строкам и по столбцам. Для обоих версий будет общая часть алгоритма:

1. В зависимости от переданных аргументов командной строки
   * Определить размеры матрица и вектора
   * Определить количество строк или столбцов на процесс
   * Сгенерировать случайную матрицу и вектор используя переданное в качестве параметра семя
2. В цикле создать заданное количество дочерних процессов
3. Каждый процесс обрабатывает свою часть матрицы
4. После создания всех дочерних процессов главный родительский процесс ждет их завершения
5. Освобождения разделяемой памяти и семафоров

Семафоры понадобятся только при разделении по столбцам т.к. при умножении одного столбца матрицы А на один и из элементов вектора b получится часть вектора c, которую необходимо прибавить к вектору c. Для обеспечения корректности эту операцию должен выполнять только один процесс в один момент времени.

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

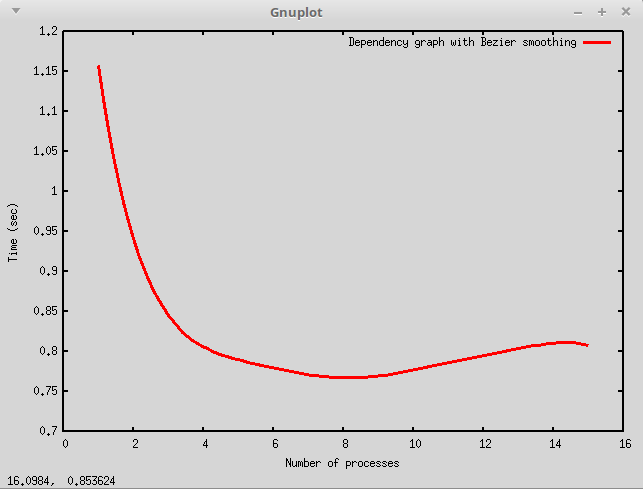
Рисунок 4 – График зависимости числа процессов от времени при разделении по строкам

Рисунок 5 – График зависимости числа процессов от времени при разделении по столбцам

Как видно из графика с ростом количества процессов время исполнения уменьшается, однако при использовании более 4 процессов 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. В приложении Г приведены таблицы зависимости времени выполнения программ от количества используемых процессов.

**Выводы**

XSI IPC позволяют организовать между процессами более сложное взаимодействия чем классические средства IPC. Более того, только с помощью XSI IPC можно организовать взаимодействие между процессами, исполняющимися в разное время.

Приложение А

rows.c

#include "sys/ipc.h"

#include "sys/shm.h"

#include "sys/sem.h"

#include "stdlib.h"

#include "stdio.h"

#include "unistd.h"

#include "sys/types.h"

#include "sys/wait.h"

#include "string.h"

#include "errno.h"

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

srand(atoi(argv[4]));

int procCount = atoi(argv[1]);

int linesCount = atoi(argv[2]);

int colsCount = atoi(argv[3]);

int linesPerProc = linesCount / procCount;

int arrayshrid = shmget(IPC\_PRIVATE, sizeof(int) \* linesCount \* colsCount, IPC\_CREAT | 0666);

int mulVectorshrid = shmget(IPC\_PRIVATE, sizeof(int) \* colsCount, IPC\_CREAT | 0666);

int resultVectorshrid = shmget(IPC\_PRIVATE, sizeof(int) \* linesCount, IPC\_CREAT | 0666);

int\* array = (int\*)shmat(arrayshrid, NULL, 0);

int\* mulVector = (int\*)shmat(mulVectorshrid, NULL, 0);

int\* resultVector = (int\*)shmat(resultVectorshrid, NULL, 0);

for (int i = 0; i < colsCount; i++) {

mulVector[i] = (-9 + rand() % 19)\*1000;

}

for (int i = 0; i < linesCount; i++) {

for (int j = 0; j < colsCount; j++) {

array[i \* colsCount + j] = (-9 + rand() % 19) \* 1000;

}

}

for (int i = 0; i < procCount; i++) {

pid\_t pid = fork();

if (pid == 0) {

int tmpResultVector = 0;

for (int j = i \* linesPerProc; j < (i + 1) \* linesPerProc; j++) {

for (int k = 0; k < colsCount; k++) {

tmpResultVector += array[i \* colsCount + k] \* mulVector[k];

}

resultVector[j] = tmpResultVector;

tmpResultVector = 0;

}

exit(0);

}

else if (pid > 0) {

continue;

}

else {

printf("Error fork");

}

}

pid\_t pid;

while (pid = waitpid(-1, NULL, 0)) {

if (errno == ECHILD) {

break;

}

}

shmctl(arrayshrid, IPC\_RMID, NULL);

shmctl(mulVectorshrid, IPC\_RMID, NULL);

shmctl(resultVectorshrid, IPC\_RMID, NULL);

return 0;

}

Приложение Б

column.c

#include "sys/ipc.h"

#include "sys/shm.h"

#include "sys/sem.h"

#include "stdlib.h"

#include "stdio.h"

#include "unistd.h"

#include "sys/types.h"

#include "sys/wait.h"

#include "string.h"

#include "errno.h"

int\* calcPart(int start, int end, int linesCount, int colsCount,

int\* mulVector, int\* array){

int\* resultVector\_ = (int\*)malloc(sizeof(int) \* linesCount);

for (int k = start; k < end; k++) {

for (int j = 0; j < linesCount; j++) {

resultVector\_[j] += array[j \* colsCount + k] \* mulVector[k];

}

}

return resultVector\_;

}

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

srand(atoi(argv[4]));

int procCount = atoi(argv[1]);

int linesCount = atoi(argv[2]);

int colsCount = atoi(argv[3]);

int linesPerProc = colsCount / procCount;

int arrayshrid = shmget(IPC\_PRIVATE, sizeof(int) \* linesCount \* colsCount, IPC\_CREAT | 0666);

int mulVectorshrid = shmget(IPC\_PRIVATE, sizeof(int) \* colsCount, IPC\_CREAT | 0666);

int resultVectorshrid = shmget(IPC\_PRIVATE, sizeof(int) \* linesCount, IPC\_CREAT | 0666);

int\* array = (int\*)shmat(arrayshrid, NULL, 0);

int\* mulVector = (int\*)shmat(mulVectorshrid, NULL, 0);

int\* resultVector = (int\*)shmat(resultVectorshrid, NULL, 0);

memset(resultVector, 0, sizeof(int) \* linesCount);

int semaforId = semget(IPC\_PRIVATE, 1, IPC\_CREAT | 0666);

struct sembuf init;

init.sem\_num = 0;

init.sem\_op = 1;

init.sem\_flg = 0;

semop(semaforId, &init, 1);

for (int i = 0; i < colsCount; i++) {

mulVector[i] = (-9 + rand() % 19) \*1000;

}

for (int i = 0; i < linesCount; i++) {

for (int j = 0; j < colsCount; j++) {

array[i \* colsCount + j] = (-9 + rand() % 19) \*1000;

}

}

for (int i = 0; i < procCount; i++) {

pid\_t pid = fork();

if (pid == 0) {

int\* resultVector\_ = (int\*)malloc(sizeof(int) \* linesCount);

memset(resultVector\_, 0, sizeof(int) \* linesCount);

resultVector\_ = calcPart(i \* linesPerProc, (i + 1) \* linesPerProc,

linesCount, colsCount, mulVector, array);

struct sembuf lock;

lock.sem\_num = 0;

lock.sem\_op = -1;

lock.sem\_flg = 0;

semop(semaforId, &lock, 1);

for (int j = 0; j < linesCount; j++) {

resultVector[j] += resultVector\_[j];

}

lock.sem\_num = 0;

lock.sem\_op = 1;

lock.sem\_flg = 0;

semop(semaforId, &lock, 1);

free(resultVector\_);

exit(0);

}

else if (pid > 0) {

continue;

}

else {

printf("Error fork");

}

}

pid\_t pid;

while (pid = waitpid(-1, NULL, 0)) {

if (errno == ECHILD) {

break;

}

}

shmctl(arrayshrid, IPC\_RMID, NULL);

shmctl(mulVectorshrid, IPC\_RMID, NULL);

shmctl(resultVectorshrid, IPC\_RMID, NULL);

semctl(semaforId, 0, IPC\_RMID, NULL);

return 0;

}

Приложение В

**run.sh**

#!/bin/bash

gcc $1

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

for (( i=1; i<=15; i++))

do

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

(time ./a.out ${i} 7000 7000 1234) &> file.txt

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

time=`head -2 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"

Приложение Г

**зависимость времени выполнения программ от количества используемых процессов**

Разделение по строкам

# Num proc Time

1 3.407

2 2.496

3 2.450

4 2.413

5 2.409

6 2.412

7 2.437

8 2.437

9 2.396

10 2.390

11 2.423

12 2.426

13 2.398

14 2.362

15 2.431

Разделение по столбцам

# Num proc Time

1 1.157

2 0.842

3 0.790

4 0.776

5 0.791

6 0.804

7 0.774

8 0.712

9 0.746

10 0.824

11 0.764

12 0.796

13 0.801

14 0.822

15 0.807