МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»

Институт информатики, математики и электроники
Факультет информатики
Кафедра технической кибернетики

Параллельные алгоритмы матричных вычислений

Отчет по лабораторной работе № 3
Вариант 2

Студенты группы 6407 Берлин Д.И.
Советников В.Р.
Проверил Головашкин Д.Л.

САМАРА 2018

СОДЕРЖАНИЕ

ЗАДАНИЕ

Согласно варианту необходимо реализовать параллельный алгоритм gaxpy по блочным строкам на процессорном кольце, так же реализовать последовательный алгоритм gaxpy, провести сравнительный анализ и построить график ускорения в зависимости от размерности матрицы.

Теоретические сведения

Операция gaxpy, есть операция вида:

$z = Ax + y$, где z,y имеет размерность $R^{n\times 1}$, x имеет размерность $R^{m\times 1}$, A есть матрица размерности $R^{n\times m}$.

Согласно варианту необходимо реализовать параллельный алгоритм строчного gaxpy.

Операция представима в виде: $z_{\mu} = \sum_{k=1}^{p} A_{\mu k} x_k + y_{\mu}$.

Запишем систолический алгоритм на процессорном кольце: Инициализация:

p – количество процессоров;

μ − номер текущего процессора;

n – размерность матрицы и векторов;

r = $\frac{n}{p}$ – количество строк в блочной строке;

left – номер процессора слева;

right – номер процессора справа;

row = (μ-1)r+1: μr;

$A_{loc} = A(row, :)$ – часть матрицы, хранимая на текущем процессоре;

$X_{loc} = x(row), Y_{loc} = y(row)$ – часть векторов хранимые, на текущем процессоре.

Конец инициализации.

Алгоритм:

for t = 1:p

send(X_{loc},right)

recv(X_{loc},left)

$\tau = \mu - t$

if $\tau \leq 0$ then $\tau = \tau + p$

$Y_{loc} = Y_{loc} + A_{loc}(:, (\tau - 1)r + 1: \tau r) X_{loc}$

end

Цель эксперимента

Целью данной работы, является получение зависимости времени от количества элементов для параллельного и последовательного алгоритма, их дальнейшая оценка в сравнении друг с другом. А также получение зависимости ускорения от размерности матрицы.

Инструментарий

Для вычислений использовалось следующее оборудование:

- для построения графиков использовался язык программирования Octave;
- язык программирования C++, компилятор gcc;
- компилятор mpicc;
- кластер с адресом sk.ssau.ru.

Данное оборудование удовлетворяет всем необходимым аппаратным и программным требованиям, т.к. оно позволяет выполнять высокопроизводительные вычисления с достаточно высокой степенью точности.

Параметры эксперимента

В качестве нижней границы матрицы была взята размерность 64x64, максимальная граница размерности матриц 1024x1024. Максимальная граница выбрана таким образом, поскольку при размерности свыше 1024 время выполнения будет серьёзно увеличиваться. А выбор минимальной

границы обусловлен тем, что при размерах матрицы меньше, чем 64x64 все алгоритмы по времени ведут себя примерно одинаково.

Язык программирования C++ выбран исходя из личных предпочтений, а так же ввиду строчного хранения матриц.

Компилятор mpicc выбран исходя из теоретического описания алгоритма, то есть потребности в пересылке данных.

Теоретическое ожидание

Ожидается, что для достаточно малых матриц время выполнения последовательной программы будет меньше, однако, с ростом этого значения, операции пересылки станут менее значимыми и параллельный алгоритм приблизится к двукратному ускорению, однако, ввиду

неидеального распараллеливания не сможет его достичь. При дальнейшем увеличении размерности, ускорение изменятся не будет.

Результаты

Результаты времени выполнения программной реализации gaxpy, на кольце при помощи технологии MPI и последовательного варианта, на языке C++ представлены на рисунке 1.

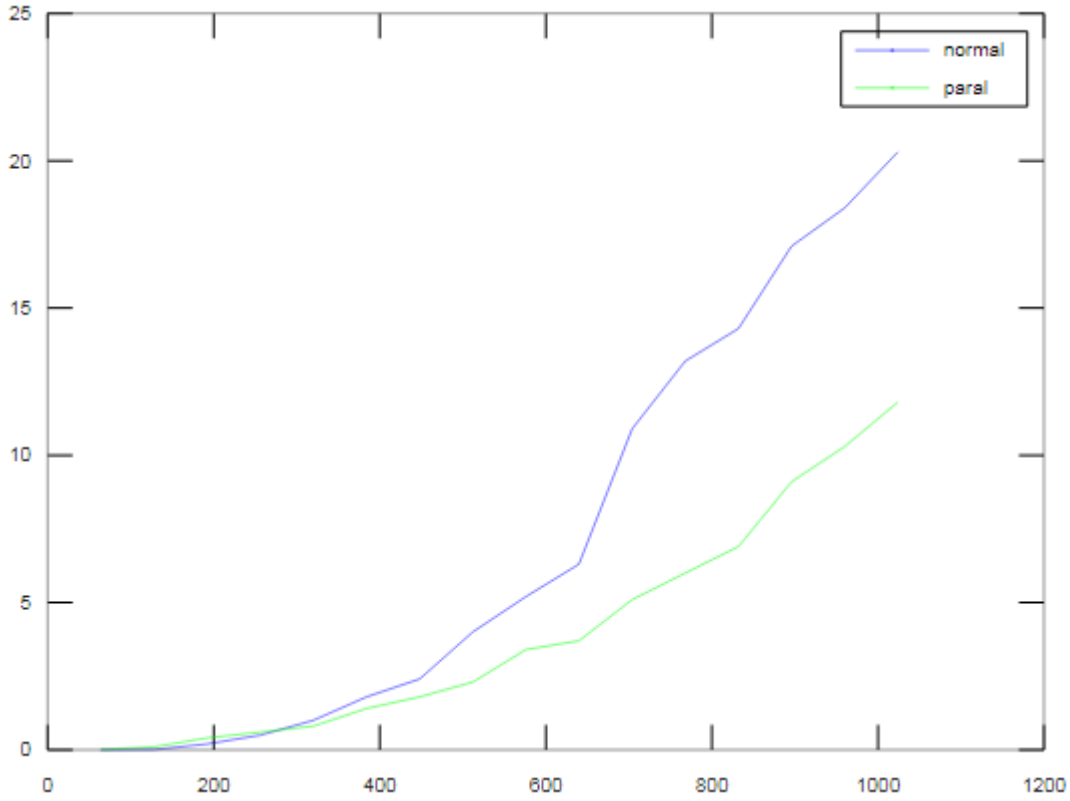

Рисунок 1 – Время работы gaxpy (время измеряется в мс)

Из рисунка видно, что параллельный вариант начинает работать быстрее с ростом размерности матрицы, в таблице 1 приведены значения времени работы обоих алгоритм.

Таблица 1 – Время работы алгоритмов

Размерность матрицы	Последовательный	Параллельный
64	0,00001	0,01
128	0,01	0,1
192	0,2	0,4
256	0,5	0,6
320	1,0	0,8
384	1,8	1,4
448	2,4	1,8
512	4,0	2,3
576	5,2	3,4
640	6,3	3,7
704	10,9	5,1
768	13,2	6,0
832	14,3	6,9
896	17,1	9,1
960	18,4	10,3
1024	20,3	11,8

На рисунке 2 так же изображено отношение последовательной реализации к параллельной.

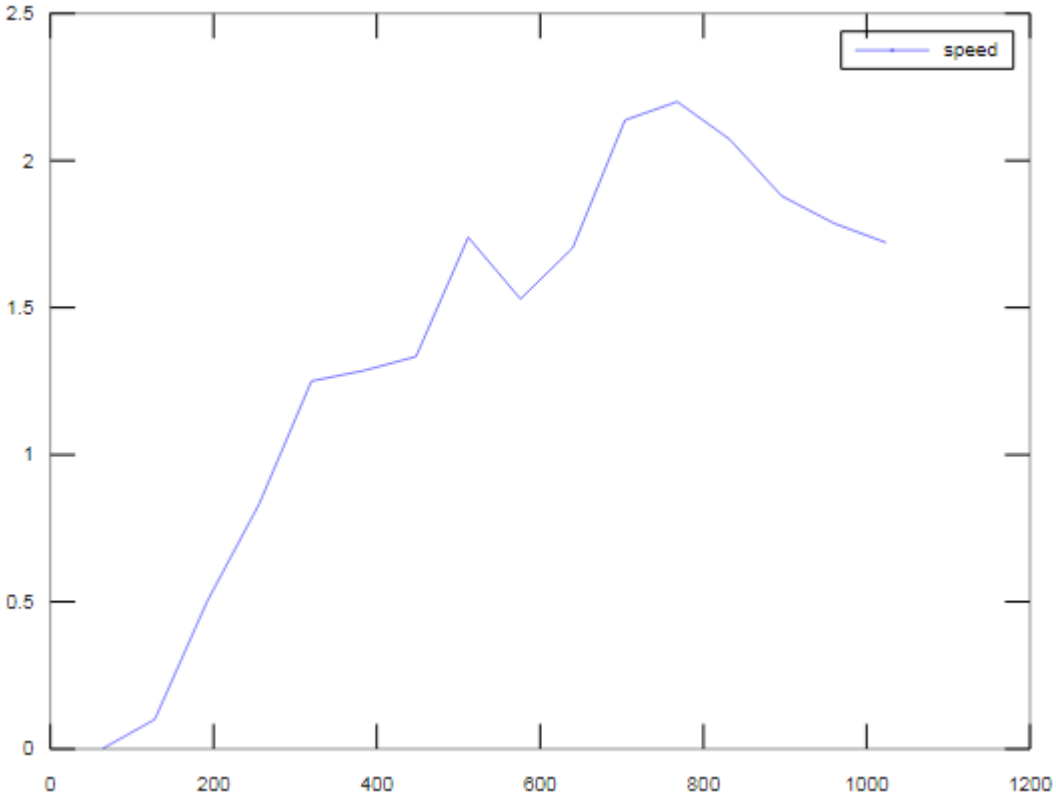

Рисунок 2 – Ускорение параллельного алгоритма

Анализ результатов

Из рисунка 2 видно, что теоретические ожидания оправдались не полностью, был промежуток, когда ускорение превышало теоретическое значение - 2. Это связано с тем, что последовательная реализация при такой размерности матрицы, сработала медленнее, поскольку для нее потребовалась большая часть памяти, в то время как параллельный алгоритм все еще был достаточен старыми размерами, увеличив значения еще сильнее, параллельному алгоритму так же потребовалась дополнительная память, и ускорение вновь стало меньше 2.

ЗАКЛЮЧЕНИЕ

В данной работе был произведен запуск последовательного алгоритма gaxpy и параллельного алгоритма gaxpy на процессорном кольце. Нам удалось получить результат, показывающий, что ускорение так же может быть связано с тем, что последовательная реализация, может сработать медленнее. Так же и отдельный процессор параллельной реализации может замедлиться по тем или иным причинам, однако поскольку он работает с меньшим количеством данных, это будет менее заметно и не сильно отразится на скорости выполнения

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Голуб Дж. Матричные вычисления / Дж. Голуб, Ч.Ван. Лоун; пер. с англ. Ю.М. Нечепуренко, А.Ю. Романова, А.В. Собянина, Е.Е. Тыртышникова; по ред. В.В. Воеводина. – М.: Мир, 1999. – 548 с
2. Вильямс Алгоритмы. Введение в разработку и анализ — М.: 2006. 189–195. — 576 с.

Приложение А
Код программы на языке C++

// Параллельный алгоритм

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <ctime>
#define NMAX 1024
int main(int argc, char** argv){
int procRank,procNum, N=NMAX, r,*row,i,j,left,right,t,T;
double start_time,end_time;
double *a_loc,*y_loc,*x_loc;
double *a = new double[NMAX*NMAX];
double *y = new double[NMAX];
double *x = new double[NMAX];
for(i=0;i<N;i++){
	for(int j=0;j<N;j++){
		a[i*N+j]=i*j;
	}
	y[i]=i;
	x[i]=i;
}
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&procNum);
MPI_Comm_rank(MPI_COMM_WORLD,&procRank);
MPI_Request request;
MPI_Request request2;
if(procRank-1<0) left=procNum-1;
else left=procRank-1;
if(procRank==procNum-1) right=0;
else right=procRank+1;
r=N/procNum;
a_loc=(double *) malloc(r*N*sizeof(double));
x_loc=(double *) malloc(r*sizeof(double));
y_loc=(double *) malloc(r*sizeof(double));
row=(int *) malloc(r*sizeof(int));
for(i=0;i<r;i++){
	row[i]=procRank*r+i;
}
for(int i=0;i<r;i++){
	y_loc[i]=y[row[i]]
	x_loc[i]=x[row[i]];
	for(j=0;j<N;j++){
		a_loc[i*N+j]=a[row[i]*N+j];
	}
}
start_time=MPI_Wtime();
for(t=0;t<procNum;t++){
	MPI_Isend(x_loc,r,MPI_DOUBLE,right,0,MPI_COMM_WORLD, &request);
	MPI_Irecv(x_loc,r,MPI_DOUBLE,left,0,MPI_COMM_WORLD, &request2);
	T=procRank-t;
	if(T<=0) T=T+procNum;
	for(i=0;i<r;i++){
		for(j=0;j<r;j++){
			y_loc[i]+=a_loc[i*N+(T-1)*r+j]*x_loc[j];
		}
	}
}
MPI_Gather(y_loc,r,MPI_DOUBLE,y,r,MPI_DOUBLE,0,MPI_COMM_WORLD);
end_time=MPI_Wtime();
end_time=end_time-start_time;
if(procRank==0){
printf("\n Time: %f\n",end_time);
}
MPI_Finalize();
return 0;
}
```

//Последовательный алгоритм

```
#include <iostream>
#include <ctime>
#include <fstream>

int main() {
	std::ofstream out;
	out.open("resultopt.txt");
	for (int n = 64; n < 1025; n = n + 64) {
		double *x = new double[n];
		double *y = new double[n];
		double st_time, end_time;
		int **a = new int*[n];
		for (int j = 0; j < n; j++) {
			a[j] = new int[n];
		}
		for (int i = 0; i < n; i++) {
			x[i] = i;
			y[i] = i;
		}
		for (int i = 0; i < n; i++) {
			for (int j = 0; j < n; j++) {
				a[i][j] = i * j;
			}
		}
		int startTime = clock();
		for (int i = 0; i < n; i++) {
			for (int j = 0; j < n; j++) {
				y[i] += a[i][j] * x[j];
			}
		}
		int endTime = clock();
		out << n << ", " << endTime - startTime << "; " << std::endl;
	}
	out.close();
}
```