Яковлев Илья М3239

группа B1

Отчет по лабораторной работе №6

Параметры системы:

- Общий объем оперативной памяти: 39,15 Гб
- Объем раздела подкачки: 10 Гб
- Размер страницы виртуальной памяти: 4096 б
- Объем свободной физической памяти в ненагруженной системе: 36,41 Гб
- Объем свободного пространства в разделе подкачки в ненагруженной системе: 9,46 Гб

Thread(s) per core:	2
CPU op-mode(s):	32-bit, 64-bit
CPU(s):	16
On-line CPU(s) list:	0-11
Off-line CPU(s) list:	12-15
CPU family:	23
Model name:	AMD Ryzen 7 2700X Eight-Core Processor
Core(s) per socket:	6

Предисловие

Рассмотрим техническое задание и посчитаем, сколько времени потребуется для сбора данных:

<u>В скобках указаны заданные в тех задании значения</u>

(5 – 10) R_OUT – сколько раз нужно повторить эксперимент

(10) R_IN – сколько раз нужно повторить запуск скрипта, описанного в 1.b, на каждой итерации цикла описанного в 1.c

(20) N_MAX – до какого значения нужно перебирать аргументы для вызова скрипта, описанного в 1.b, на каждой итерации цикла описанного в 1.c

(2,5 сек) T – среднее время работы функции из условия

Потребуется провести 8 экспериментов, 4 из которых с последовательным вызовом, другие 4 с параллельным вызовом. Будем считать, что первая группа работает за сумму времен работы каждой ее строчки, а вторая группа работает в 2 раза быстрее.

Тогда получаем формулу общего времени сбора измерений в секундах:

(4*1 + 4*1/2) * T * R_OUT * R_IN * N_MAX * (N_MAX + 1)/2 =

= 3 * T * R_OUT * R_IN* N_MAX * (N_MAX + 1)

Для данных нам значений она равна 315’000 секундам, а это примерно 3,5 суток непрерывных вычислений!

Причем не учитываются накладные расходы для, например, обновления файлов для эксперимента с большими объемами данных.

Так что становится понятно, что требования в тех задании соблюсти крайне сложно, из-за чего в моих опытах используются другие значения, указанные ниже:

R_OUT = 4

R_IN = 3

N_MAX = 6

T = 1 секунда

Считаю, что, используя данные значения, можно получить репрезентативный результат.

Замечу, что время работы каждого из 6 экспериментов составило 15-20 минут.

Описание проведения экспериментов

Описание изучаемых алгоритмов:

exp_1_hard_eval.sh

```
#!/bin/bash

N=250000
MODULE=1000000007
x=$1

for ((i = 0; i < $N; i++))
do
  x=$((x * x % MODULE))
done

echo $x
```

Данный скрипт вычисляет (x ^ (2 ^ N)) % MODULE

x принимается в качестве параметра.

N подобрано так, чтобы соответствовать выбранному ранее T.

exp_2_big_data_eval.sh

```
#!/bin/bash

ind=$(($1 - 2))
DATA_DIR=exp_2_dataset
DATA_PATH=$DATA_DIR/"data_$ind"

cat $DATA_PATH |
while read val
do
  if [[ $val == "END" ]]
  then
    break
  fi
  val=$((val * 2))
  echo $val
done >> $DATA_PATH
```

Данный скрипт на каждой итерации берет число из файла, умножает его на 2 и добавляет полученное значение в конец файла.

Имя файла получено с помощью параметра $1.

create_random_data.sh

```
#!/bin/bash

#250 KB
SIZE=50000
N=6

for (( nxt=0; nxt < $N; nxt++ ))
do
    for (( i=0; i < SIZE; i++ ))
    do
        echo $RANDOM
    done > /home/tedes/lab6/exp_2_dataset/"data_$nxt"

    echo "END" >> /home/tedes/lab6/exp_2_dataset/"data_$nxt"
done
```

Скрипт для генерации случайных файлов, используемых в качестве данных для exp_2_big_data_eval.sh

start_seq.sh

```
#!/bin/bash

for ((i = 2; i <= $2 + 1; i++))
do
  ./$1 $i
done > /dev/null
```

Данный скрипт принимает в качестве первого аргумента имя скрипта, а в качестве второго аргумента сколько раз $1 нужно последовательно исполнить.

start_par.sh

```
#!/bin/bash

for ((i = 2; i <= $2 + 1; i++))
do
  ./$1 $i &
done > /dev/null

temp=$(ps r | grep "./$1")
while [ -n "$temp" ]
do
    sleep 0.3
    temp=$(ps r | grep "./$1")
done
```

Аналогично предыдущему скрипту, но запускает $1 в фоновом режиме $2 раз.

Скрипт не завершается, пока не завершатся все запущенные $1.

Стоит заметить, что это ожидание немного сказывается на итоговую оценку времени, увеличивая его.

Как установил экспериментальный подбор sleep 0.3 влияет меньше всего:

Если уменьшить значение, то слишком часто start_par.sh будет потреблять процессорное время, замедляя общее выполнение.

Если увеличить значение, то будет слишком большой отклик на окончание работы ожидаемых процессов.

start_general.sh

```
#!/bin/bash

N_MAX=6
REPEAT_CNT=3

touch .tmp

for ((N = 1; N <= $N_MAX; N++))
do
  sum=0
  for ((i = 0; i < $REPEAT_CNT; i++))
  do
    ./$3
    sudo time -f"%e" -ao .tmp ./$1 $2 $N
  done
  sum=$(cat .tmp | awk '{s+=$1}END{print s}')
  avg=$(awk -v x=$sum -v n=$REPEAT_CNT 'BEGIN { print x / n }')
  echo "$N $avg"
  >.tmp
done

sudo rm .tmp
```

Рассмотрим данный скрипт:

$1 – имя скрипта, время работы которого будет замерено, причем ему в параметры передастся $2 и $N (переменная принимающая значения 1 – N_MAX)

$3 – имя скрипта, который будет запускаться перед замером времени $1

Варьируя $1 и $2 можно провести все эксперименты, которые нам потребуются.

В качестве $1 передаются starter’ы последовательного или параллельного вычисления, в качестве $2 анализируемые скрипты.

В качестве $3 при exp_1_hard_eval.sh передается NOP, который ничего не делает, а при exp_2_big_data_eval.sh передается create_random_data.sh, который обновляет испорченный на предыдущем шаге dataset.

start_exp_1_part_1.sh

```
#!/bin/bash

./start_general.sh start_seq.sh exp_1_hard_eval.sh do_nothing.sh
```

Запускает эксперимент для exp_1_hard_eval.sh с последовательным вызовом.

start_exp_1_part_2.sh

```
#!/bin/bash

./start_general.sh start_par.sh exp_1_hard_eval.sh do_nothing.sh
```

Запускает эксперимент для exp_1_hard_eval.sh с параллельным вызовом.

start_exp_2_part_1.sh

```
#!/bin/bash

./start_general.sh start_seq.sh exp_2_big_data_eval.sh ./exp_2_dataset/create_random_data.sh
```

Запускает эксперимент для exp_2_big_data_eval.sh с последовательным вызовом.

start_exp_2_part_2.sh

```
#!/bin/bash

./start_general.sh start_par.sh exp_2_big_data_eval.sh exp_2_dataset/create_random_data.sh
```

Запускает эксперимент для exp_2_big_data_eval.sh с параллельным вызовом.

repeat.sh

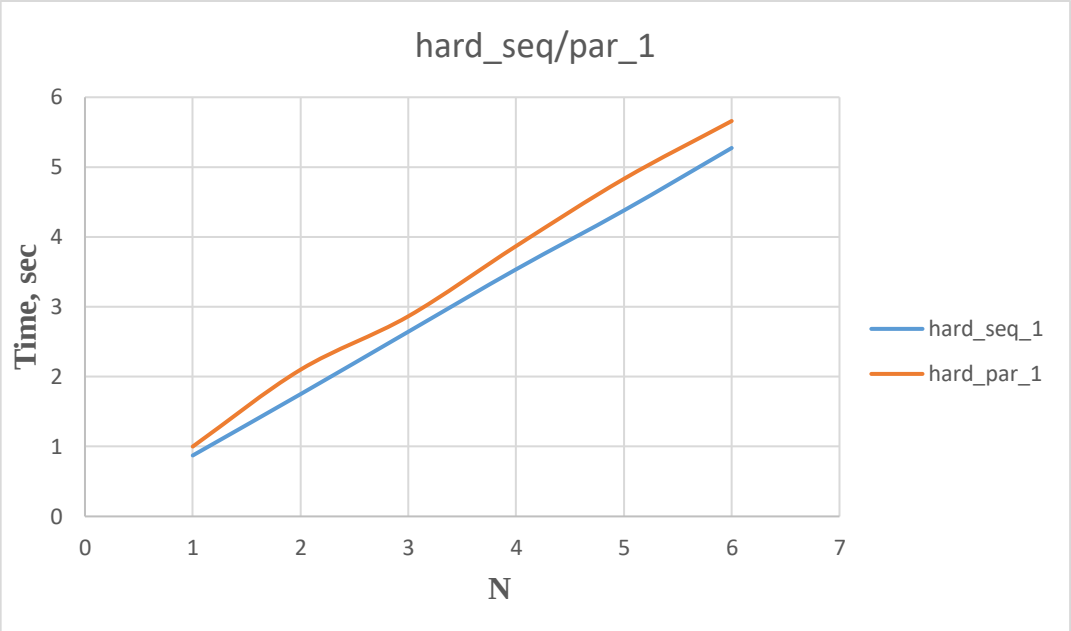
```
#!/bin/bash

REPEAT_CNT=4

./"$1" > .repeat

for ((i = 1; i < $REPEAT_CNT; i++))
do
  ./"$1" | awk '{print $2}' > .repeat.tmp
  if [ -f .repeat ]
  then
    paste .repeat .repeat.tmp | awk '{print $1, $2 + $3}' > q
    mv q .repeat
  else
    mv .repeat.tmp .repeat
  fi
done

cat .repeat | awk -v rep_cnt=$REPEAT_CNT '{print $1, $2 / rep_cnt}'
```

Данный скрипт запускает несколько раз скрипт $1 и берет среднее по всем вызовам

Графики и их анализ:

Введем обозначения для экспериментов:

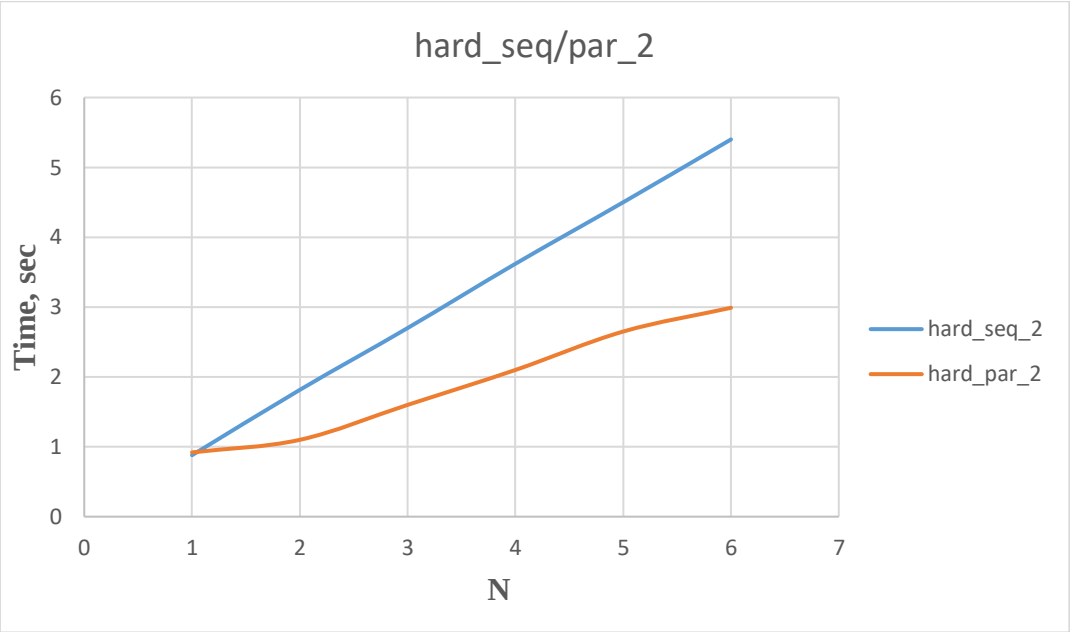
- hard / big_data – изучение exp_1_hard_eval.sh или exp_2_big_data_eval.sh соответсвенно
- seq / par – изучение с последовательным или параллельным вызовом соответственно
- 1 / 2 – число используемых процессоров в изучении



Видим линейный рост с одинаковым наклоном у обоих графиков, но у параллельного вызова всегда немного больше время работы.

Вероятно, это из-за небольшого замедления start_par.sh, о котором было сказано ранее.

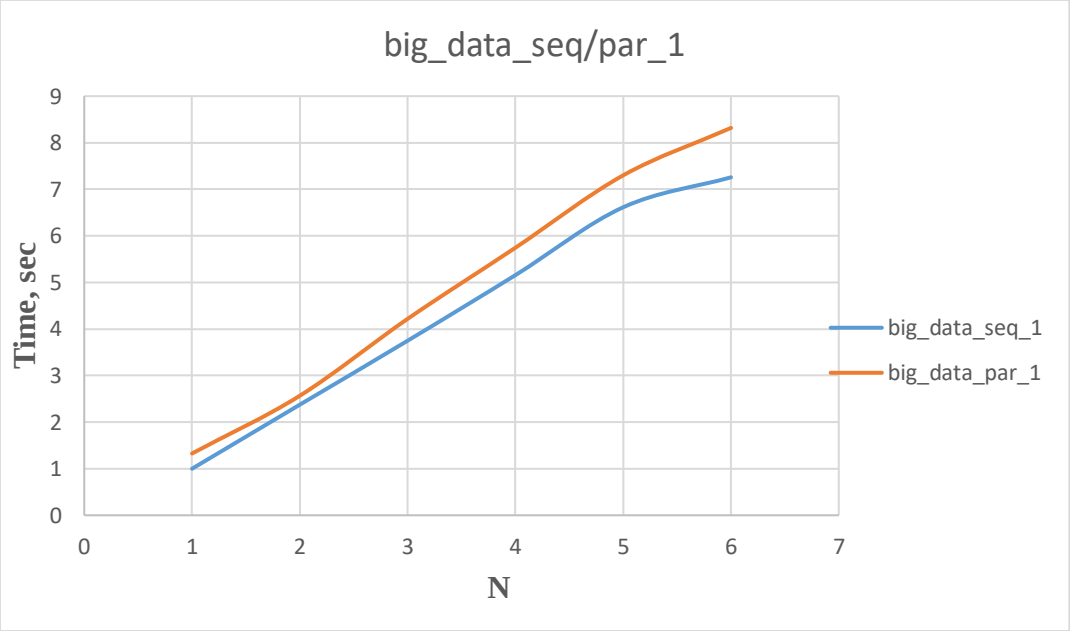
Полученный результат вполне ожидаем, ведь раз процессор 1, вид исполнения не должен влиять на время.



Опять же, видим линейный рост, но у последовательного исполнения рост быстрее.

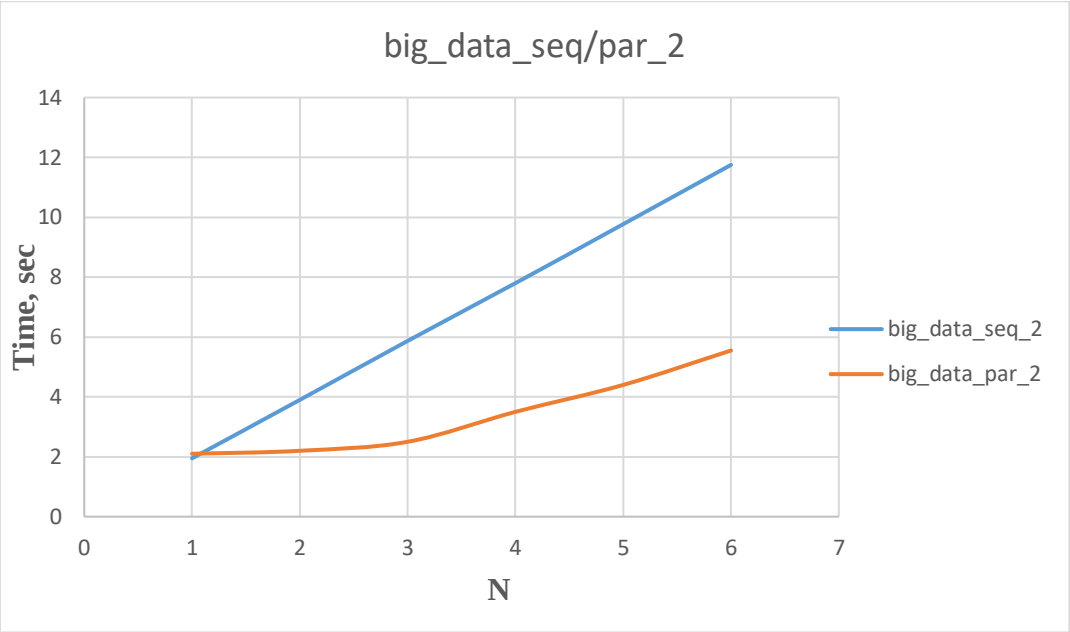
Причем на отрезку N = [1;2] почти нет роста у параллельного исполнения, так как второй процессор начинает обрабатывать второй вызов exp_1_hard_eval.sh

Результат ожидаем: 2 процессора дают выигрыш при параллельном исполнении почти в 2 раза.



Рост линеен, растут с одинаковой скоростью.

Причины были оговорены выше.



Рост последовательного исполнения линеен, параллельного начиная с N=3 становится линеен, а до этого почти нет роста.

Причины были оговорены выше.

Стоит заметить, что для этого графика с 2 процессами разница роста значительно сильнее, чем в предыдущем. Полагаю, это связано с тем, что exp_2_big_data_eval.sh не делает трудных вычислений и его каждая итерация хорошо разделена на чтение, вычисление и изменение, из-за чего он лучше распараллеливается.

Итог

Резюмируем то, что было сказано выше:

- Как бы не было странно, нет смысла искать выигрыш в скорости в распараллеливании процессов, если у вас 1 процессор.
- Больший выигрыш от оптимизации с распараллеливанием получают скрипты, которые оперируют с разными, не пересекающимися, данными, делают несложные операции, т е требуют не слишком много процессорного времени.
- Выигрыш от распараллеливания особенно высок, если количество скриптов, которые мы хотим вызвать, примерно равно количеству процессоров.