Parallel Processing

#5 Homework Solutions

محمد حسین خوشه چین - ۹۹۲۱۰۱۶۴

۱۴ خرداد ۱۴۰۰

مشخصات سیستم

سیستم عامل بنده Pop-OS می باشد که یک توزیع از سیستم عامل Ubuntu است. مشخصات پردازنده سیستم بنده در شکل ۱ قابل مشاهده است. همانطور که در شکل قابل مشاهده است

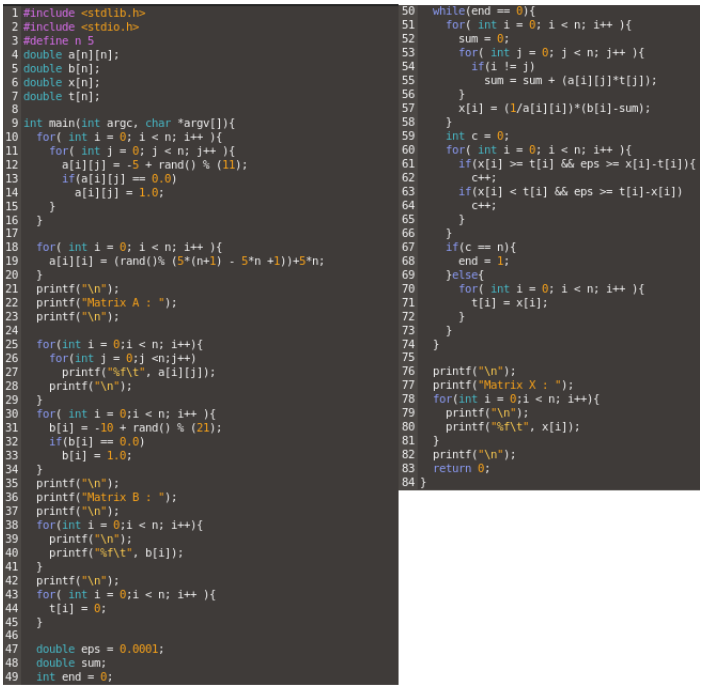
```
kc@pop-os:~$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   36 bits physical, 48 bits virtual
CPU(s):                          4
On-line CPU(s) list:             0-3
Thread(s) per core:              2
Core(s) per socket:              2
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           58
Model name:                      Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz
Stepping:                        9
CPU MHz:                         1712.489
CPU max MHz:                     3100.0000
CPU min MHz:                     1200.0000
BogoMIPS:                        4988.91
Virtualization:                  VT-x
L1d cache:                       64 KiB
L1i cache:                       64 KiB
L2 cache:                       512 KiB
L3 cache:                       3 MiB
NUMA node0 CPU(s):               0-3
Vulnerability Itlb multihit:     KVM: Mitigation: VMX disabled
```

شکل ۱

تعداد socket های سیستم بنده برابر با ۱ می باشد و در هر socket به تعداد ۲ عدد core وجود دارد و هر core میتواند ۲ thread داشته باشد. بنابراین سیستم میتواند تا ۴ thread در سطح kernel space داشته باشد.

مسئله متد Jacobi

در ابتدا یک پیاده سازی ترتیبی از الگوریتم Jacobi که در صفحه ویکی پدیا معرفی شده است با زبان برنامه نویسی C انجام می دهیم. پیاده سازی این الگوریتم در شکل ۲ آمده است.



```c
#include <stdlib.h>
#include <stdio.h>
#define n 5
double a[n][n];
double b[n];
double x[n];
double t[n];

int main(int argc, char *argv[]){
  for( int i = 0; i < n; i++ ){
    for( int j = 0; j < n; j++ ){
      a[i][j] = -5 + rand() % (11);
      if(a[i][j] == 0.0)
        a[i][j] = 1.0;
    }
  }

  for( int i = 0; i < n; i++ ){
    a[i][i] = (rand()% (5*(n+1) - 5*n +1))+5*n;
  }
  printf("\n");
  printf("Matrix A : ");
  printf("\n");

  for(int i = 0;i < n; i++){
    for(int j = 0;j <n;j++)
      printf("%f\t", a[i][j]);
    printf("\n");
  }
  for( int i = 0;i < n; i++ ){
    b[i] = -10 + rand() % (21);
    if(b[i] == 0.0)
      b[i] = 1.0;
  }
  printf("\n");
  printf("Matrix B : ");
  printf("\n");
  for(int i = 0;i < n; i++){
    printf("\n");
    printf("%f\t", b[i]);
  }
  printf("\n");
  for( int i = 0;i < n; i++ ){
    t[i] = 0;
  }

  double eps = 0.0001;
  double sum;
  int end = 0;
  while(end == 0){
    for( int i = 0; i < n; i++ ){
      sum = 0;
      for( int j = 0; j < n; j++ ){
        if(i != j)
          sum = sum + (a[i][j]*t[j]);
      }
      x[i] = (1/a[i][i])*(b[i]-sum);
    }
    int c = 0;
    for( int i = 0; i < n; i++ ){
      if(x[i] >= t[i] && eps >= x[i]-t[i]){
        c++;
      if(x[i] < t[i] && eps >= t[i]-x[i])
        c++;
      }
    }
    if(c == n){
      end = 1;
    }else{
      for( int i = 0; i < n; i++ ){
        t[i] = x[i];
      }
    }
  }

  printf("\n");
  printf("Matrix X : ");
  for(int i = 0;i < n; i++){
    printf("\n");
    printf("%f\t", x[i]);
  }
  printf("\n");
  return 0;
}
```

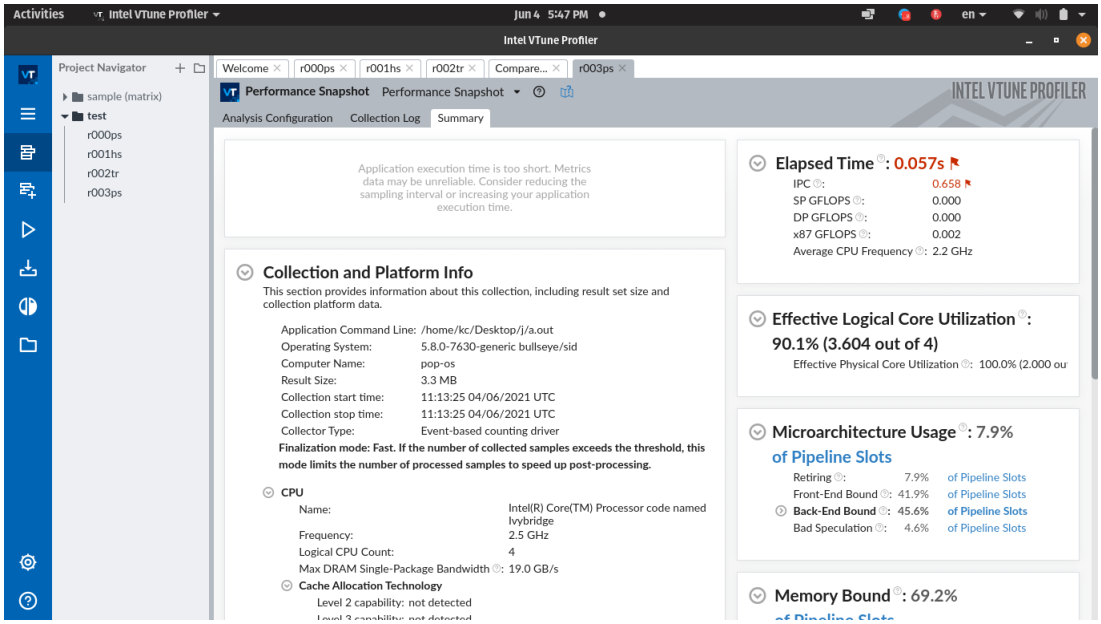
شکل ۲

یک نمونه از اجرای این برنامه را می توانید در شکل ۳ مشاهده کنید. حال فایل قبل اجرای

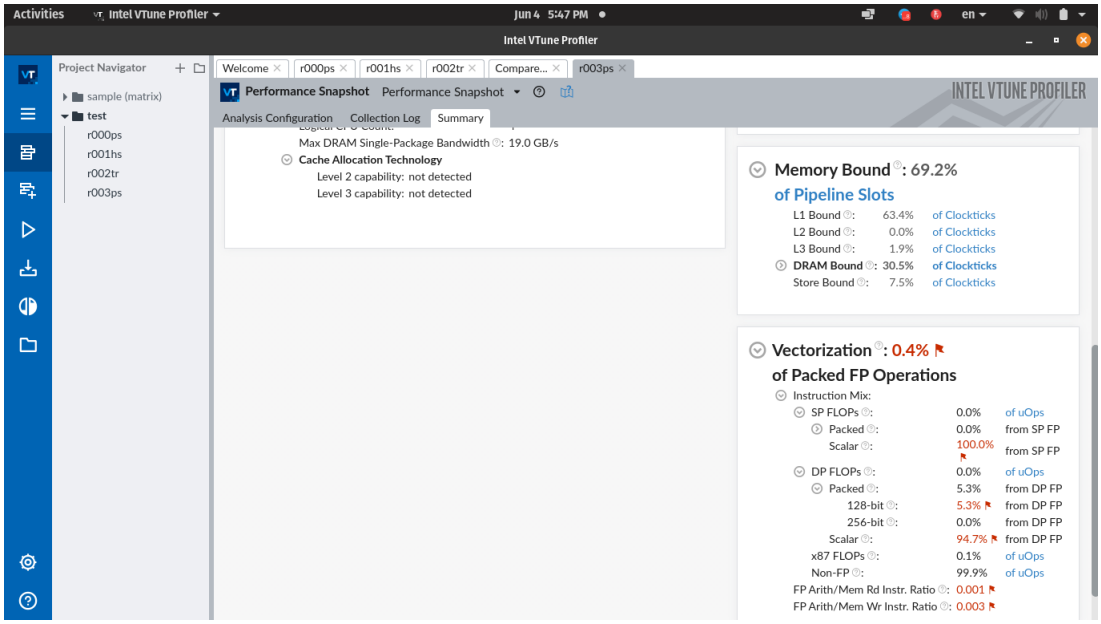
```
kc@pop-os:~/Desktop/j$ gcc sequentialJacobi.c
kc@pop-os:~/Desktop/j$ ./a.out

Matrix A :
27.000000       5.000000        1.000000        -3.000000       -4.000000
-1.000000       27.000000       1.000000        -2.000000       -4.000000
3.000000        2.000000        26.000000       -2.000000       2.000000
-1.000000       4.000000        5.000000        26.000000       -5.000000
5.000000        3.000000        1.000000        -5.000000       26.000000

Matrix B :

-5.000000
-10.000000
-7.000000
2.000000
7.000000

Matrix X :
-0.041221
-0.290597
-0.251920
0.238939
0.366327
```

شکل ۳

برنامه مان را به برنامه Intel Vtune Profiler می دهیم تا تحلیل کارایی کد ای که نوشتیم را به ما نشان دهد. خروجی این مرحله را می توانید در شکل ۴ و ۵ مشاهده کنید.



شکل ۴



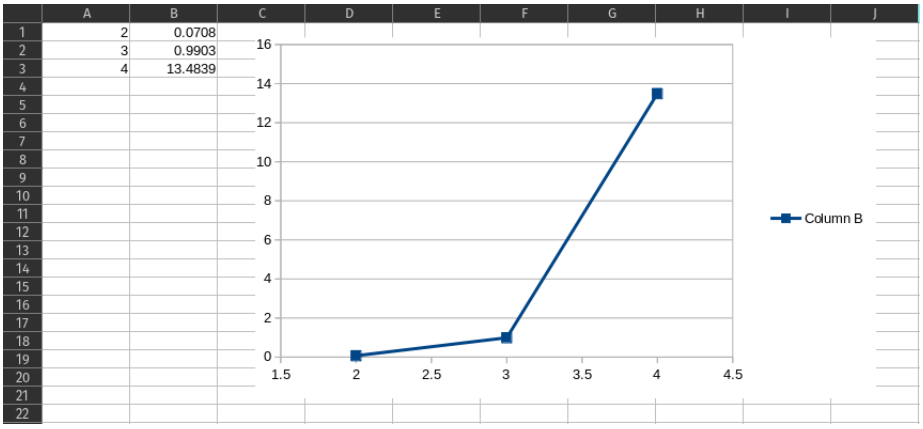
شکل ۵

حال با کمک OpenMP برنامه ترتیبی ای که برای مسئله ژاکوبی نوشته بودیم را به یک نسخه موازی شده تبدیل می کنیم. نسخه موازی شده برنامه را می توانید در شکل ۶ مشاهده کنید.

```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#define n 10000
double a[n][n];
double b[n];
double x[n];
double t[n];
int main(int argc, char *argv[]){
  int i,j,c;
  double start_time, stop_time;
  start_time = omp_get_wtime();
  omp_set_num_threads(5);
#pragma omp parallel private ( i , j )
{
#pragma omp for
  for( i = 0; i < n; i++ ){
    for( j = 0; j < n; j++ ){
      a[i][j] = -5 + rand() % (11);
      if(a[i][j] == 0.0)
        a[i][j] = 1.0;
    }
  }
#pragma omp for
  for( i = 0; i < n; i++ ){
    a[i][i] = (rand()% (5*(n+1) - 5*n +1))+5*n;
  }
#pragma omp for
  for( i = 0;i < n; i++ ){
    b[i] = -10 + rand() % (21);
    if(b[i] == 0.0)
      b[i] = 1.0;
  }
#pragma omp for
  for( i = 0;i < n; i++ ){
    t[i] = 0;
  }
}
  printf("\n");
  printf("Matrix A : ");
  printf("\n");
  for( i = 0;i < n; i++){
    for( j = 0;j <n;j++)
      printf("%f\t", a[i][j]);
    printf("\n");
  }
  printf("\n");
  printf("Matrix B : ");
  printf("\n");
  for( i = 0;i < n; i++){
    printf("\n");
    printf("%f\t", b[i]);
  }
  printf("\n");

  double eps = 0.0001;
  double sum;
  int end = 0;
  while(end == 0){
#pragma omp parallel shared(k) private ( i , j , sum)
{
#pragma omp for
    for(  i = 0; i < n; i++ ){
      sum = 0;
      for(  j = 0; j < n; j++ ){
        if(i != j)
          sum = sum + (a[i][j]*t[j]);
      }
      x[i] = (1/a[i][i])*(b[i]-sum);
    }
    c = 0;
#pragma omp for reduction( +:c )
    for(  i = 0; i < n; i++ ){
      if(x[i] >= t[i] && eps >= x[i]-t[i]){
        c = c+1;
      if(x[i] < t[i] && eps >= t[i]-x[i])
        c = c+1;
      }
    }
    if(c == n){
      end = 1;
    }else{
#pragma omp for
      for(  i = 0; i < n; i++ ){
        t[i] = x[i];
      }
    }
}
  }
  printf("\n");
  printf("Matrix X : ");
  for( i = 0;i < n; i++){
    printf("\n");
    printf("%f\t", x[i]);
  }
  printf("\n");
  stop_time = omp_get_wtime();
  double run_time = stop_time - start_time;
  printf("\n Execution time was %lf seconds\n ",run_time);
  printf("\n");
  return 0;
}
```

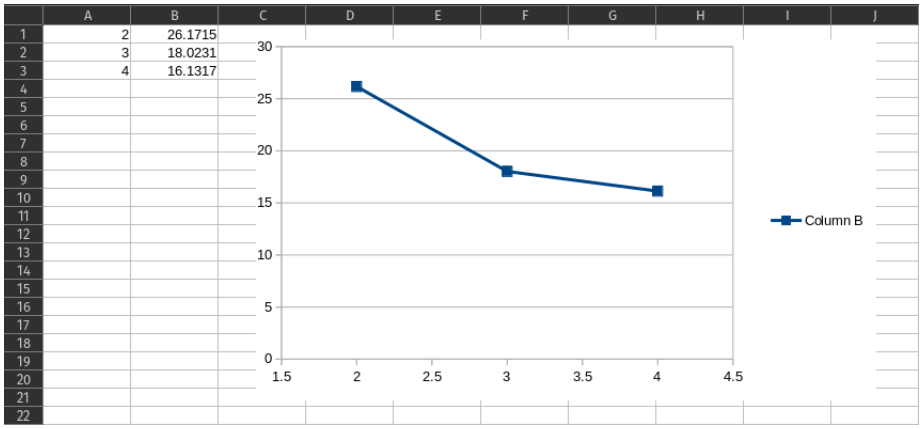
شکل ۶

حال برای تست این برنامه را با تعداد ورودی ۱۰ یعنی n=10 و به ترتیب با ۲ و ۳ و ۴ thread اجرا می کنیم. و زمان اجرای آن را که در خروجی چاپ می شود را در قالب یک نمودار ارائه می کنیم. این نمودار را در شکل ۷ مشاهده می کنید.



شکل ۷

همانطور که ملاحظه می کنید به ازای ورودی های کوچک موازی سازی سربار زیادی ایجاد میکند و باعث افزایش یافتن زمان اجرای برنامه می شود. حال برنامه را با ورودی ۱۰۰۰۰ یعنی n=10000 و به ترتیب با ۲ و ۳ و ۴ thread اجرا می کنیم. و زمان اجرای آن را که در خروجی چاپ می شود را در قالب یک نمودار ارائه می کنیم. این نمودار را در شکل ۸ مشاهده می کنید. همانطور که ملاحظه می کنید با افزایش ورودی موازی سازی برنامه بهینه تر خواهد



شکل ۸

بود.

مسئله quicksort

با توجه به الگوریتم ای که در صفحه ویکی پدیا این مبحث معرفی شده است ابتدا یک نسخه ترتیبی و بازگشتی از این الگوریتم را با زبان C می نویسیم و سپس با استفاده از OpenMP آن را موازی سازی می کنیم. نسخه موازی سازی شده این الگوریتم در شکل ۹ آمده است.

```c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#define n 10000000
int data[n];

int partition(int data[],int s,int f){
  int index = s-1;
  int pivot = data[f];
  for(int i=s ; i<=f-1; i++){
    if(data[i] < pivot){
       index++;
       int temp = data[index];
       data[index] = data[i];
       data[i] = temp;
    }
  }
  int temp = data[index+1];
  data[index+1] = data[f];
  data[f] = temp;
  return (index+1);
}

void quicksort(int data[],int s,int f){
  int p;
  if(s < f){
    p = partition(data,s,f);
    #pragma omp task default(none) firstprivate(data,s,p)
    {
      quicksort(data,s,p-1);
    }
    #pragma omp task default(none) firstprivate(data,f,p)
    {
      quicksort(data,p+1,f);
    }
  }
}

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

  double start_time, stop_time;
  for( int i = 0; i < n; i++ ){
    data[i] = rand() % 1000 +1;
  }
  omp_set_num_threads(4);
  start_time = omp_get_wtime();
  #pragma omp parallel default(none) shared(data)
  {
    #pragma omp single nowait
    {
      quicksort(data,0,n-1);
    }
  }
  stop_time = omp_get_wtime();

  double run_time = stop_time - start_time;
  printf("\n Execution time was %lf seconds\n ",run_time);
  printf("\n");
  return 0;
}
```

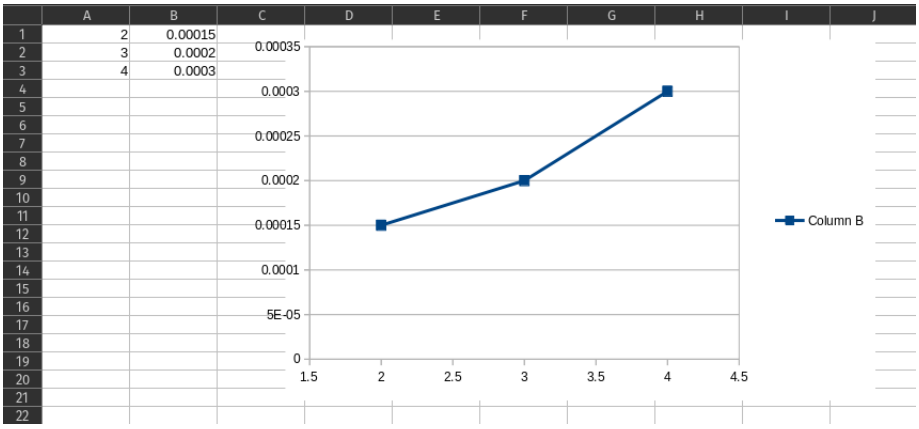
شکل ۹

به عنوان نمونه این برنامه را برای ۱۰ عدد ورودی اجرا می کنیم. خروجی برنامه در شکل ۱۰ آمده است.

```
kc@pop-os:~/Desktop/q$ cc sort.c -fopenmp
kc@pop-os:~/Desktop/q$ ./a.out

Input :
384     887     778     916     794     336     387     493     650     422

Input :
336     384     387     422     493     650     778     794     887     916

 Execution time was 0.000140 seconds

kc@pop-os:~/Desktop/q$
```

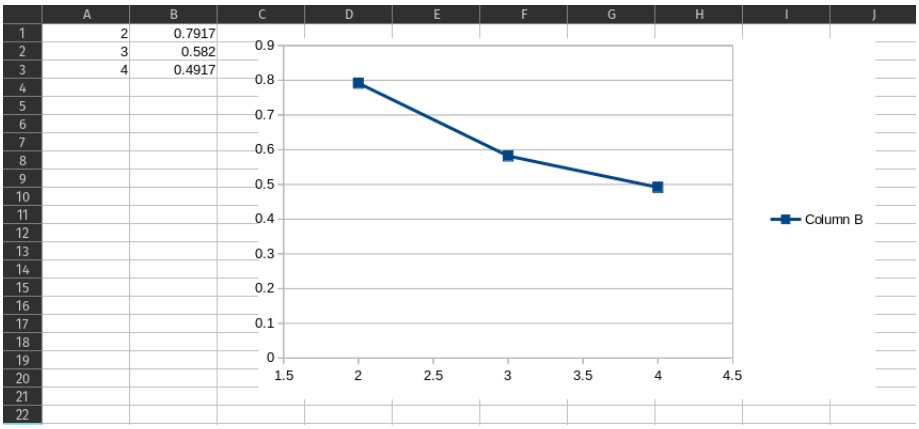
شکل ۱۰

حال برای تست این برنامه را با تعداد ورودی ۱۰ یعنی n=10 و به ترتیب با ۲ و ۳ و ۴ thread اجرا می کنیم. و زمان اجرای آن را که در خروجی چاپ می شود را در قالب یک نمودار ارائه می کنیم. این نمودار را در شکل ۱۱ مشاهده می کنید.



شکل ۱۱

همانطور که ملاحظه می کنید به ازای ورودی های کوچک موازی سازی سربار زیادی ایجاد میکند و باعث افزایش یافتن زمان اجرای برنامه می شود. حال برنامه را با ورودی ۱۰۰۰۰۰۰ یعنی n=1000000 و به ترتیب با ۲ و ۳ و ۴ thread اجرا می کنیم. و زمان اجرای آن را که در خروجی چاپ می شود را در قالب یک نمودار ارائه می کنیم. این نمودار را در شکل ۱۲ مشاهده می کنید. همانطور که ملاحظه می کنید با افزایش ورودی موازی سازی برنامه بهینه تر



شکل ۱۲

خواهد بود.

مراجع

[1] A. M. Andrew. Another Efficient Algorithm for Conex Hulls in Two Dimensions, Info. Proc. Letters9, 216-219, 1979.