PARALLEL AND DISTRIBUTED COMPUTING LAB

REPORT

NAME: S Shyam Sundaram

REG NO: 19BCE1560

PROGRAMMING ENVIRONMENT: OpenMP

PROBLEM: Profiling

DATE: 6th October, 2021

HARDWARE CONFIGURATION:

CPU NAME	:	Intel core i5 – 1035G1 @ 1.00 Ghz
Number of Sockets:	:	1
Cores per Socket	:	4
Threads per core	:	1
L1 Cache size	:	320KB
L2 Cache size	:	2MB
L3 Cache size (Shared):		6MB
RAM	:	8 GB

STATEMENT

The sorting algorithm, Bubble Sort, is implemented serially and in a parallelized manner using OpenMP as functions. The program is then profiled using functional, line and hardware profiling techniques.

CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

#define SIZE 10000

int* initArray(int r,int u,int l)
{
   srand(time(0));

   int *arr=(int*)malloc(r*sizeof(int));

   for(int j=0;j<r;++j)
      arr[j]=(rand()%(u-l+1))+l;
```

```
   return arr;
}

void freeArray(int* arr)
{
   free(arr);
}

void bubbleSortSerial(int a[])
{
   int temp;
   for(int i=0;i<SIZE;++i)
   {
      for(int j=i;j<SIZE;++j)
      {
         if(a[i]>a[j+1])
         {
            temp=a[j];
            a[j]=a[j+1];
            a[j+1]=temp;
         }
      }
   }
}

void bubbleSortParallel(int a[])
{
Int k;
   omp_set_num_threads(4);
   for(int i=0;i<SIZE;++i)
   {
      k=i%2;
      #pragma omp paralel for default(none) shared(k,a)
      for(int j=k;j<SIZE-1;j+=2)
      {
         if(a[j]>a[j+1])
         {
            int temp=a[j];
            a[j]=a[j+1];
            a[j+1]=temp;
         }
      }
```

```
    }
}

void main()
{
    int *a=initArray(SIZE,100,0);
    int *b=initArray(SIZE,100,0);

    float start=omp_get_wtime();
    bubbleSortSerial(a);
    float end=omp_get_wtime();
    float exec=end-start;
    printf("Time taken is: %f\n",exec);
    freeArray(a);

    start=omp_get_wtime();
    bubbleSortParallel(b);
    end=omp_get_wtime();
    exec=end-start;
    printf("Time taken is: %f\n",exec);

    freeArray(b);
}
```

FUNCTIONAL PROFILING

COMPILATION AND EXECUTION

gcc -fopenmp -pg prog.c -o prog
./prog
gprof prog
gprof -b prog

SCREENSHOTS

```
shyam@shyam-Inspiron-14-5408:~/Academics/Lab-Fall-2021/PDC/Lab8$ gcc -fopenmp -pg prog.c -o prog
shyam@shyam-Inspiron-14-5408:~/Academics/Lab-Fall-2021/PDC/Lab8$ ./prog
Time taken is: 1.205078
Time taken is: 0.778809
```

Flat profile:

```
shyam@shyam-Inspiron-14-5408:~/Academics/Lab-Fall-2021/PDC/Lab8$ gprof prog
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 57.30      1.13     1.13        1     1.13     1.13  bubbleSortSerial
 39.73      1.92     0.79        1     0.79     0.79  bubbleSortParallel
  0.00      1.92     0.00        2     0.00     0.00  freeArray
  0.00      1.92     0.00        2     0.00     0.00  initArray

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.


Copyright (C) 2012-2020 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
```

Call Graph:

```
                     Call graph (explanation follows)


 granularity: each sample hit covers 2 byte(s) for 0.52% of 1.92 seconds

 index % time    self  children    called     name
                                                <spontaneous>
 [1]    100.0    0.00    1.92                 main [1]
                 1.13    0.00       1/1           bubbleSortSerial [2]
                 0.79    0.00       1/1           bubbleSortParallel [3]
                 0.00    0.00       2/2           initArray [5]
                 0.00    0.00       2/2           freeArray [4]
 -----------------------------------------------
                 1.13    0.00       1/1           main [1]
 [2]     59.1    1.13    0.00       1         bubbleSortSerial [2]
 -----------------------------------------------
                 0.79    0.00       1/1           main [1]
 [3]     40.9    0.79    0.00       1         bubbleSortParallel [3]
 -----------------------------------------------
                 0.00    0.00       2/2           main [1]
 [4]      0.0    0.00    0.00       2         freeArray [4]
 -----------------------------------------------
                 0.00    0.00       2/2           main [1]
 [5]      0.0    0.00    0.00       2         initArray [5]
 -----------------------------------------------

  This table describes the call tree of the program, and was sorted by
  the total amount of time spent in each function and its children.

  Each entry in this table consists of several lines.  The line with the
  index number at the left hand margin lists the current function.
  The lines above it list the functions that called this function,
  and the lines below it list the functions this one called.
  This line lists:
      index      A unique number given to each element of the table.
                 Index numbers are sorted numerically.
                 The index number is printed next to every function name so
                 it is easier to look up where the function is in the table.
```

```
                 included in the number after the `/'.

     name        This is the name of the parent.  The parent's index
                 number is printed after it.  If the parent is a
                 member of a cycle, the cycle number is printed between
                 the name and the index number.

  If the parents of the function cannot be determined, the word
  `<spontaneous>' is printed in the `name' field, and all the other
  fields are blank.

  For the function's children, the fields have the following meanings:

     self        This is the amount of time that was propagated directly
                 from the child into the function.

     children    This is the amount of time that was propagated from the
                 child's children to the function.

     called      This is the number of times the function called
                 this child `/' the total number of times the child
                 was called.  Recursive calls by the child are not
                 listed in the number after the `/'.

     name        This is the name of the child.  The child's index
                 number is printed after it.  If the child is a
                 member of a cycle, the cycle number is printed
                 between the name and the index number.

  If there are any cycles (circles) in the call graph, there is an
  entry for the cycle-as-a-whole.  This entry shows who called the
  cycle (as parents) and the members of the cycle (as children.)
  The `+' recursive calls entry shows the number of function calls that
  were internal to the cycle, and the calls entry for each member shows,
  for that member, how many times it was called from other members of
  the cycle.


 Copyright (C) 2012-2020 Free Software Foundation, Inc.

 Copying and distribution of this file, with or without modification,
 are permitted in any medium without royalty provided the copyright
 notice and this notice are preserved.


 Index by function name

    [3] bubbleSortParallel      [4] freeArray
    [2] bubbleSortSerial        [5] initArray
```

GPROF with brief report of flat profile and call graph: (gprof -b prog)

```
shyam@shyam-Inspiron-14-5408:~/Academics/Lab-Fall-2021/PDC/Lab8$ gprof -b prog
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 57.30      1.13     1.13        1     1.13     1.13  bubbleSortSerial
 39.73      1.92     0.79        1     0.79     0.79  bubbleSortParallel
  0.00      1.92     0.00        2     0.00     0.00  freeArray
  0.00      1.92     0.00        2     0.00     0.00  initArray


                     Call graph


granularity: each sample hit covers 2 byte(s) for 0.52% of 1.92 seconds

index % time    self  children    called     name
                                               <spontaneous>
[1]    100.0    0.00    1.92                 main [1]
                1.13    0.00       1/1           bubbleSortSerial [2]
                0.79    0.00       1/1           bubbleSortParallel [3]
                0.00    0.00       2/2           initArray [5]
                0.00    0.00       2/2           freeArray [4]
-----------------------------------------------
                1.13    0.00       1/1           main [1]
[2]     59.1    1.13    0.00       1         bubbleSortSerial [2]
-----------------------------------------------
                0.79    0.00       1/1           main [1]
[3]     40.9    0.79    0.00       1         bubbleSortParallel [3]
-----------------------------------------------
                0.00    0.00       2/2           main [1]
[4]      0.0    0.00    0.00       2         freeArray [4]
-----------------------------------------------
                0.00    0.00       2/2           main [1]
[5]      0.0    0.00    0.00       2         initArray [5]
-----------------------------------------------


Index by function name

   [3] bubbleSortParallel      [4] freeArray
   [2] bubbleSortSerial        [5] initArray
shyam@shyam-Inspiron-14-5408:~/Academics/Lab-Fall-2021/PDC/Lab8$
```

INFERENCES AND OBSERVATIONS

From the profile generated, we see that serial bubble sort function takes the longest time. It is followed by parallel bubble sort and free and array initialising functions take very little time.

LINE PROFILING

COMPILATION AND EXECUTION

gcc -fopenmp -fprofile-arcs -ftest-coverage prog.c -o prog
./prog
gcov prog.c
cat prog.c.gcov

SCREENSHOTS

```
shyam@shyam-Inspiron-14-5408:~/Academics/Lab-Fall-2021/PDC/Lab8$ cat prog.c.gcov
        -:    0:Source:prog.c
        -:    0:Graph:prog.gcno
        -:    0:Data:prog.gcda
        -:    0:Runs:1
        -:    1:#include <stdio.h>
        -:    2:#include <stdlib.h>
        -:    3:#include <time.h>
        -:    4:#include <omp.h>
        -:    5:
        -:    6:#define SIZE 10000
        -:    7:
        2:    8:int* initArray(int r,int u,int l)
        -:    9:{
        2:   10:    srand(time(0));
        -:   11:
        2:   12:    int *arr=(int*)malloc(r*sizeof(int));
        -:   13:
    20002:   14:    for(int j=0;j<r;++j)
    20000:   15:        arr[j]=(rand()%(u-l+1))+l;
        -:   16:
        -:   17:
        2:   18:    return arr;
        -:   19:}
        -:   20:
        2:   21:void freeArray(int* arr)
        -:   22:{
        2:   23:    free(arr);
        2:   24:}
        -:   25:
        1:   26:void bubbleSortSerial(int a[])
        -:   27:{
        -:   28:    int temp;
    10001:   29:    for(int i=0;i<SIZE;++i)
        -:   30:    {
 50015000:   31:        for(int j=i;j<SIZE;++j)
        -:   32:        {
 50005000:   33:            if(a[i]>a[j+1])
        -:   34:            {
 12175751:   35:                temp=a[j];
 12175751:   36:                a[j]=a[j+1];
 12175751:   37:                a[j+1]=temp;
        -:   38:            }
        -:   39:        }
        -:   40:    }
        1:   41:}
        -:   42:
        1:   43:void bubbleSortParallel(int a[])
        -:   44:{
```

```
        1:   43:void bubbleSortParallel(int a[])
        -:   44:{
    10001:   45:    for(int i=0;i<SIZE;++i)
        -:   46:    {
    10000:   47:        int first=i%2;
        -:   48:        #pragma omp paralel for default(none) shared(a,first)
 50005000:   49:        for(int j=first;j<SIZE-1;j+=2)
        -:   50:        {
 49995000:   51:            if(a[j]>a[j+1])
        -:   52:            {
 24336714:   53:                int temp=a[j];
 24336714:   54:                a[j]=a[j+1];
 24336714:   55:                a[j+1]=temp;
        -:   56:            }
        -:   57:        }
        -:   58:    }
        1:   59:}
        -:   60:
        1:   61:void main()
        -:   62:{
        1:   63:    int *a=initArray(SIZE,100,0);
        1:   64:    int *b=initArray(SIZE,100,0);
        -:   65:
        1:   66:    float start=omp_get_wtime();
        1:   67:    bubbleSortSerial(a);
        1:   68:    float end=omp_get_wtime();
        1:   69:    float exec=end-start;
        1:   70:    printf("Time taken is: %f\n",exec);
        1:   71:    freeArray(a);
        -:   72:
        1:   73:    start=omp_get_wtime();
        1:   74:    bubbleSortParallel(b);
        1:   75:    end=omp_get_wtime();
        1:   76:    exec=end-start;
        1:   77:    printf("Time taken is: %f\n",exec);
        1:   78:    freeArray(b);
        1:   79:}
shyam@shyam-Inspiron-14-5408:~/Academics/Lab-Fall-2021/PDC/Lab8$
```

Write individual execution counts for every basic block and branch frequencies to the output file along with branch summary info: (gcov -b -a prog.c)

```
shyam@shyam-Inspiron-14-5408:~/Academics/Lab-Fall-2021/PDC/Lab8$ cat prog.c.gcov
        -:    0:Source:prog.c
        -:    0:Graph:prog.gcno
        -:    0:Data:prog.gcda
        -:    0:Runs:1
        -:    1:#include <stdio.h>
        -:    2:#include <stdlib.h>
        -:    3:#include <time.h>
        -:    4:#include <omp.h>
        -:    5:
        -:    6:#define SIZE 10000
        -:    7:
function initArray called 2 returned 100% blocks executed 100%
        2:    8:int* initArray(int r,int u,int l)
        -:    9:{
        2:   10:    srand(time(0));
        2:   10-block  0
call    0 returned 100%
call    1 returned 100%
        -:   11:
        2:   12:    int *arr=(int*)malloc(r*sizeof(int));
        -:   13:
    20002:   14:    for(int j=0;j<r;++j)
    20002:   14-block  0
branch  0 taken 100%
branch  1 taken 1% (fallthrough)
    20000:   15:        arr[j]=(rand()%(u-l+1))+l;
    20000:   15-block  0
call    0 returned 100%
        -:   16:
        -:   17:
        2:   18:    return arr;
        2:   18-block  0
        -:   19:}
        -:   20:
function freeArray called 2 returned 100% blocks executed 100%
        2:   21:void freeArray(int* arr)
        -:   22:{
        2:   23:    free(arr);
        2:   24:}
        -:   25:
function bubbleSortSerial called 1 returned 100% blocks executed 100%
        1:   26:void bubbleSortSerial(int a[])
        -:   27:{
        -:   28:    int temp;
    10001:   29:    for(int i=0;i<SIZE;++i)
```

```
 50005000:   33:            if(a[i]>a[j+1])
 50005000:   33-block  0
branch  0 taken 24% (fallthrough)
branch  1 taken 76%
        -:   34:            {
 12175751:   35:                temp=a[j];
 12175751:   36:                a[j]=a[j+1];
 12175751:   37:                a[j+1]=temp;
 12175751:   37-block  0
        -:   38:            }
        -:   39:        }
        -:   40:    }
        1:   41:}
        -:   42:
function bubbleSortParallel called 1 returned 100% blocks executed 100%
        1:   43:void bubbleSortParallel(int a[])
        -:   44:{
    10001:   45:    for(int i=0;i<SIZE;++i)
        1:   45-block  0
    10000:   45-block  1
    10001:   45-block  2
branch  0 taken 100%
branch  1 taken 1% (fallthrough)
        -:   46:    {
    10000:   47:        int first=i%2;
        -:   48:        #pragma omp paralel for default(none) shared(a,first)
 50005000:   49:        for(int j=first;j<SIZE-1;j+=2)
    10000:   49-block  0
 49995000:   49-block  1
 50005000:   49-block  2
branch  0 taken 100%
branch  1 taken 1% (fallthrough)
        -:   50:        {
 49995000:   51:            if(a[j]>a[j+1])
 49995000:   51-block  0
branch  0 taken 49% (fallthrough)
branch  1 taken 51%
        -:   52:            {
 24336714:   53:                int temp=a[j];
 24336714:   54:                a[j]=a[j+1];
 24336714:   55:                a[j+1]=temp;
 24336714:   55-block  0
        -:   56:            }
        -:   57:        }
        -:   58:    }
        1:   59:}
        -:   60:
function main called 1 returned 100% blocks executed 100%
        1:   61:void main()
        -:   62:{
```

Use colors for lines of code that have zero coverage. (gcov -k prog.c)

```
shyam@shyam-Inspiron-14-5408:~/Academics/Lab-Fall-2021/PDC/Lab8$ cat prog.c.gcov
        -:    0:Colorization: profile count: zero coverage (exceptional) zero coverage (unexceptional) unexecuted block
        -:    0:Source:prog.c
        -:    0:Graph:prog.gcno
        -:    0:Data:prog.gcda
        -:    0:Runs:1
        -:    1:#include <stdio.h>
        -:    2:#include <stdlib.h>
        -:    3:#include <time.h>
        -:    4:#include <omp.h>
        -:    5:
        -:    6:#define SIZE 10000
        -:    7:
        2:    8:int* initArray(int r,int u,int l)
        -:    9:{
        2:   10:    srand(time(0));
        -:   11:
        2:   12:    int *arr=(int*)malloc(r*sizeof(int));
        -:   13:
    20002:   14:    for(int j=0;j<r;++j)
    20000:   15:        arr[j]=(rand()%(u-l+1))+l;
        -:   16:
        -:   17:
        2:   18:    return arr;
        -:   19:}
        -:   20:
```

Output summaries for each function in addition to the file level summary (gcov -f prog.c)

```
shyam@shyam-Inspiron-14-5408:~/Academics/Lab-Fall-2021/PDC/Lab8$ gcov -f prog.c
Function 'main'
Lines executed:100.00% of 16

Function 'bubbleSortParallel'
Lines executed:100.00% of 10

Function 'bubbleSortSerial'
Lines executed:100.00% of 8

Function 'freeArray'
Lines executed:100.00% of 3

Function 'initArray'
Lines executed:100.00% of 6

File 'prog.c'
Lines executed:100.00% of 43
Creating 'prog.c.gcov'

shyam@shyam-Inspiron-14-5408:~/Academics/Lab-Fall-2021/PDC/Lab8$
```

INFERENCES AND OBSERVATIONS

Gcov helps us profile our code line by line. Various options of gcov are also explored such as coloring lines with no coverage, display summaries of each function etc.

HARDWARE PROFILING

COMPILATION AND EXECUTION

gcc -fopenmp prog.c -o prog
./prog
likwid-topology
likwid-pin -c 0,1,2,3 ./prog

SCREENSHOTS

```
shyam@shyam-Inspiron-14-5408:~/Academics/Lab-Fall-2021/PDC/Lab8$ likwid-topology
--------------------------------------------------------------------------------
CPU name:       Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz
CPU type:       Intel Icelake processor
CPU stepping:   5
********************************************************************************
Hardware Thread Topology
********************************************************************************
Sockets:                1
Cores per socket:       4
Threads per core:       2
--------------------------------------------------------------------------------
HWThread        Thread          Core            Socket          Available
0               0               0               0               *
1               0               1               0               *
2               0               2               0               *
3               0               3               0               *
4               1               0               0               *
5               1               1               0               *
6               1               2               0               *
7               1               3               0               *
--------------------------------------------------------------------------------
Socket 0:               ( 0 4 1 5 2 6 3 7 )
--------------------------------------------------------------------------------
********************************************************************************
Cache Topology
********************************************************************************
Level:                  1
Size:                   48 kB
Cache groups:           ( 0 4 ) ( 1 5 ) ( 2 6 ) ( 3 7 )
--------------------------------------------------------------------------------
Level:                  2
Size:                   512 kB
Cache groups:           ( 0 4 ) ( 1 5 ) ( 2 6 ) ( 3 7 )
--------------------------------------------------------------------------------
Level:                  3
Size:                   6 MB
Cache groups:           ( 0 4 1 5 2 6 3 7 )
--------------------------------------------------------------------------------
********************************************************************************
NUMA Topology
********************************************************************************
NUMA domains:           1
--------------------------------------------------------------------------------
Domain:                 0
Processors:             ( 0 1 2 3 4 5 6 7 )
Distances:              10
Free memory:            2224.02 MB
Total memory:           7723.62 MB
--------------------------------------------------------------------------------
```

```
shyam@shyam-Inspiron-14-5408:~/Academics/Lab-Fall-2021/PDC/Lab8$ gcc -fopenmp prog.c -o prog
shyam@shyam-Inspiron-14-5408:~/Academics/Lab-Fall-2021/PDC/Lab8$ ./prog
Time taken is: 1.187988
Time taken is: 0.778809
shyam@shyam-Inspiron-14-5408:~/Academics/Lab-Fall-2021/PDC/Lab8$ likwid-pin -c 0,1,2,3 ./prog
Sleeping longer as likwid_sleep() called without prior initialization
Time taken is: 1.162598
Time taken is: 0.783691
```

INFERENCES AND OBSERVATIONS

Likwid is a suite of command line applications used to estimate a program's performance. With 'likwid-topology', we are shown the thread, cache, NUMA and GPU topologies, properties and information. With likwid-pin, we pin each software thread to hardware and evaluate the code. This means we specify which threads to use (as seen in third screenshot command and output).

CONCLUSION

We have used three different profiling techniques and tools to evaluate the performance of serial and parallel bubble sort code written in C. Various options of gprof and gcov were also explored.