Assignment Description

1. OpenMP fundamentals

The first part of this work consists of understanding how to program and execute simple OpenMP programs.

Our first program is "hello world" style, as shown below:

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

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

#pragma omp parallel private(nthreads, tid)
  {

  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  }
}
```

Please note that this example follows the basic OpenMP syntax:

```
#include “omp.h” int main ()
{
     int var1, var2, var3;
    // Serial code
    ...
    // Beginning of parallel section.
    // Fork a team of threads. Specify variable scoping
    #pragma omp parallel private(var1, var2) shared(var3)
    {
       // Parallel section executed by all threads
       ...
       // All threads join master thread and disband
    }
    // Resume serial code ...
}
```

2. Executing OpenMP

The code shown above can be compiled with the following option:

```
gcc -fopenmp hello_omp.c -o hello_omp
```

It is expected that the binary you obtained from the compilation is executed **using SGE**; however, the following example shows how to run an OpenMP program with different numbers of OpenMP filters.

```
[ivan@eimtarqso]$ gcc -fopenmp hello_omp.c -o hello_omp

[ivan@eimtarqso]$ ./hello_omp
Hello World from thread = 2
Hello World from thread = 1
Hello World from thread = 3
Hello World from thread = 0
Number of threads = 4
##(by default OpenMP uses: OpenMP threads = CPU cores)

[ivan@eimtarqso]$ export OMP_NUM_THREADS=3
##(we specify the number of threads to be used with the environment variable)

[ivan@eimtarqso]$ ./hello_omp
Hello World from thread = 1
Hello World from thread = 0
Number of threads = 3
Hello World from thread = 2

[ivan@eimtarqso]$ export OMP_NUM_THREADS=2

[ivan@eimtarqso]$ ./hello_omp
Hello World from thread = 1
Hello World from thread = 0
Number of threads = 2
```

Please note that the number of OpenMP threads can be specified inside of the OpenMP code. However, in general, OpenMP applications rely on the variable OMP_NUM_THREADS to specify the level of parallelism (i.e., number of threads to use).

The following scripts depict two possible ways to run a program using OpenMP through SGE:

```
##(omp1.sge)

#!/bin/bash
#$ -cwd
#$ -S /bin/bash
#$ -N omp1
#$ -o omp1.out.$JOB_ID
#$ -e omp1.out.$JOB_ID
#$ -pe openmp 3

export OMP_NUM_THREADS=$NSLOTS
./hello_omp
```

In the previous example, we defined the number of CPU cores that will be assigned to the job through "**#$ -pe openmp 3**" and we use $NSLOTS (which takes the value provided to "OpenMP -pe") to specify the number of OpenMP threads.

```
##(omp2.sge)

#!/bin/bash
#$ -cwd
#$ -S /bin/bash
#$ -N omp1
#$ -o omp1.out.$JOB_ID
#$ -e omp1.out.$JOB_ID
#$ -pe openmp 3

./hello_omp
```

This second example assumes that the variable OMP_NUM_THREADS is defined externally. The following command can be used to submit the task:

```
[ivan@eimtarqso]$ qsub -v OMP_NUM_THREADS='3' h.sge
```

Questions:

1. **How many OpenMP threads you would use in the UOC cluster (hint: use lscpu command) ? Why? Elaborate an inconvenient of using other configurations.**

3. Data parallelism

The example below shows the use of "parallel" and "for" clauses in two different but equivalent ways.

```
#pragma omp parallel
{
     #pragma omp for
          for(i=0; i<N; i++){
               c[i] = b[i]+a[i];
          }
}

##(we will target this second way moving forward)

#pragma omp parallel for
     for(i=0; i<N; i++){
          c[i] = b[i]+a[i];
     }
```

The previous example shows the sum of two vectors.

Questions:

2. **How you would implement a program that does the sum of the different elements stored in a vector "a" (sum=a[0]+…+a[N-1]) using the reduction clause?**

4. Functional parallelism

In this exercise you can also use task parallelism, if necessary (not mandatory). OpenMP also supports tasks. Differently from what data parallelism does, where the same operation is done over all the input elements, task parallelism enables executing different parts of the code to the input data. An OpenMP task typically has the code to execute, a data set, and an thread associated with the code and data set.

The following code illustrates the use of OpenMP sections (see more details in the materials provided on the UOC campus).

```
#pragma omp parallel shared(n,a,b,c,d) private(i)
{
     #pragma omp sections nowait
          #pragma omp section
               for (i=0; i<n; i++)
                    d[i] = 1.0/c[i];
          #pragma omp section
               for (i=0; i<n-1; i++)
                    b[i] = (a[i] + a[i+1])/2;
     } /*-- End of sections --*/
} /*-- End of parallel region
```

Note that this code will run in parallel two different loops (sequentially) on different arrays using an OpenMP thread for each of them.

5. Performance evaluation (OpenMP)

We will use an illustrative example based on a matrix multiplication without any optimizations. You will be provided, along with this document, with two different programs that implement matrix multiplication (mm.c and mm2.c). The difference between these two is that in the second version we have exchanged the indexes of the outer and inner loops as shown below:

mm.c

```
  (...)
  for (i=0;i<SIZE;i++)
   for(j=0;j<SIZE;j++)
    for(k=0;k<SIZE;k++)
      mresult[i][j]=mresult[i][j] + matrixa[i][k]*matrixb[k][j];
  (...)
```

mm2.c

```
(...)
for (k=0;k<SIZE;k++)
 for(j=0;j<SIZE;j++)
  for(i=0;i<SIZE;i++)
    mresult[i][j]=mresult[i][j] + matrixa[i][k]*matrixb[k][j];
(...)
```

To compile these examples just use:

```
gcc mm.c -o mm
```

With the default matrix size (SIZE) equal to 1000 you will get a fast execution time but a time that will allow you to observe differences between the two versions of the matrix multiplication.

Using the command `time`, (for example, `time ./mm`) you will be able to observe a noticeable difference in the execution time.

Note:

Remember that you are expected to make your executions through the queuing system, if you do not do so you may have totally random performance degradation due to the sharing of resources (for example due to memory containment).

It is clear that there is an impact on the performance of the multiplication execution by modifying the loop indices and, therefore, how we are accessing the data in memory. Therefore, in this case, the time system tool is not enough to understand (or check quantitatively) this situation.

Note (2):

Do not optimize the compilation of the provided code (do NOT use options like -O3 for example) since the automatic optimizations will reduce the execution times and will make typical modifications (alignment, loop unrolling, etc.) that won't allow you to follow this part of the PAC. Remember that we are working on an illustrative example.

6. OpenMP scheduling policies

In this section of the PAC you are asked to implement the parallel version of the provided programs ("mm.c" and "mm2.c") using OpenMP.

Once the parallelization is done, a performance study has to be performed with variations of the following parameters:

- Size of the matrix (at least 4 different sizes)
- Number of threads (1,..,4 that are the number of threads available in a node)
- Scheduling policy (static, dynamic and guided)

At least the following data has to be provided:

- Execution time for the sequential version of the program for each of the matrix sizes.
- Execution time for the parallel version of the program for each of the matrix sizes, for each of the level of parallelism and for each of the scheduling policies.
- Speedup observed
- Other performance studies that the student may find interesting.

Questions:

3. **Provide the parallel version of mm.c and mm2.c codes using the different scheduling policies. Please describe your main implementation decisions.**
4. **Provide the performance evaluation described above. Elaborate the results obtained.**

7. Profiling Applications

In order to study the behavior of these programs and the impact on the use of resources, we will use PAPI (Performance Application Programming Interface). PAPI is a very useful interface that allows you to use counter hardware that is available in most modern microprocessors.

We provide you with a couple of examples of how to use PAPI but it is expected that you will do your own search for information and examples. You can start through the following online link:

http://icl.cs.utk.edu/papi/.

PAPI is available in the UOC cluster in the following directories:

/export/apps/papi/ (login node)

/share/apps/papi/ (compute nodes)

In the example flops.c and flops2.c provided with this PAC, an example of PAPI usage is shown. These examples use a PAPI interface that provides the MFLOPS obtained in the execution of the program. It also provides a more accurate run time and processor time.

To compile them you can use the following command:

```
gcc -I/export/apps/papi/include/ flops.c /export/apps/papi/lib/libpapi.a -o flops
# Note that all programs that use PAPI must #include papi.h
```

You will observe a significant difference in the FLOPS obtained for the two versions of the multiplication of matrices.

Questions:

5. **Why do you think it is this difference in execution time and MFLOPs?**

In the latest version of the examples provided (counters.c and counters2.c) hardware counters are used explicitly. To compile them you only have to do as before:

```
gcc -I/export/apps/papi/include/ counters.c /export/apps/papi/lib/libpapi.a -o counters
```

You will see that the output of the execution of these examples gives you the total amount of instructions (PAPI_TOT_INS) and floating point operations (PAPI_FP_OPS).

You can observe that the output indicates that the two examples are executing the same number of floating point operations (although you can see a certain variability in the total number of instructions). Clearly, other resources related to memory access must be analyzed to better understand the impact of the exchange of loop indexes.

In this PAC we ask you to use other hardware counters to study the examples provided. You can check the meaning of the counters used in the examples of the PAC, the total set of hardware counters available to the processors of the UOC cluster, and the available events by using the following command in the login node:

```
/export/apps/papi/bin/papi_avail
```

You will obtain the following result (partial - the output is cut off):

```
[@eimtarqso]$ /export/apps/papi/bin/papi_avail
Available PAPI preset and user defined events plus hardware information.
--------------------------------------------------------------------------------
PAPI Version             : 5.5.0.0
Vendor string and code   : GenuineIntel (1)
Model string and code    : Intel(R) Xeon(R) CPU           E5603  @ 1.60GHz (44)
CPU Revision             : 2.000000
CPUID Info               : Family: 6  Model: 44  Stepping: 2
CPU Max Megahertz        : 1600
CPU Min Megahertz        : 1200
Hdw Threads per core     : 1
Cores per Socket         : 4
Sockets                  : 1
NUMA Nodes               : 1
CPUs per Node            : 4
Total CPUs               : 4
Running in a VM          : no
Number Hardware Counters : 7
Max Multiplex Counters   : 32
--------------------------------------------------------------------------------
================================================================================
  PAPI Preset Events
================================================================================
    Name        Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   Yes  Level 2 data cache misses
PAPI_L2_ICM  0x80000003  Yes   No   Level 2 instruction cache misses
PAPI_L3_DCM  0x80000004  No    No   Level 3 data cache misses
PAPI_L3_ICM  0x80000005  No    No   Level 3 instruction cache misses
PAPI_L1_TCM  0x80000006  Yes   Yes  Level 1 cache misses
PAPI_L2_TCM  0x80000007  Yes   No   Level 2 cache misses
PAPI_L3_TCM  0x80000008  Yes   No   Level 3 cache misses
PAPI_CA_SNP  0x80000009  No    No   Requests for a snoop
PAPI_CA_SHR  0x8000000a  No    No   Requests for exclusive access to shared cache line
PAPI_CA_CLN  0x8000000b  No    No   Requests for exclusive access to clean cache line
PAPI_CA_INV  0x8000000c  No    No   Requests for cache line invalidation
PAPI_CA_ITV  0x8000000d  No    No   Requests for cache line intervention
PAPI_L3_LDM  0x8000000e  Yes   No   Level 3 load misses
PAPI_L3_STM  0x8000000f  No    No   Level 3 store misses
(...)
--------------------------------------------------------------------------------
Of 108 possible events, 58 are available, of which 14 are derived.

avail.c                                 PASSED
```

Questions:

6. **What hardware counters would you use to study the differences between the two implementations? Why?**
7. **Provide the code where you use hardware counters to quantify the differences between the two examples. Provide the results obtained.**
8. **Provide parallel versions (OpenMP) and the results obtained using PAPI for the two examples.**

8. Parallelization problem with OpenMP

The provided file p2.c implements a problem sequentially. The result (output) could be shown using `printf` but we will only focus on functionality of the program using OpenMP.

Questions:

9. **Provide a parallel implementation of p2.c using OpenMP.**

Grading Policy

The correct use of the OpenMP programming model will be evaluated. The correct programming, structure and operation of the programs and the discussion of the performance evaluation will also be taken into account.

Submission Format and Due Date

Please submit your assignment as a single PDF containing all the responses including the scripts, graphs etc. Brevity and concretion will be positively considered to grade the assignment.

Submitting a tar or zip file including files with your implementation is allowed (not preferred), but please do NOT use rar.

Assignments should be submitted by **22 April 2020**.