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:

```
##(omp.sge)

#!/bin/bash
#$ -cwd
#$ -S /bin/bash
#$ -N omp
#$ -o omp.out.$JOB_ID
#$ -e omp.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.

```
#!/bin/bash
#$ -cwd
#$ -S /bin/bash
#$ -N omp
#$ -o omp.out.$JOB_ID
#$ -e omp.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
```

Q1 (0.25/10). How many OpenMP threads you would use in the UOC cluster (hint: use can use the lscpu command) ? Explain why and 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 in the subsequent examples)

#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.

Q2 (0.75/10). Provide a parallel implementation (OpenMP) of 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. Memory access in 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

gcc mm2.c -o mm2
```

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) 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 assignment. Remember that we are working on an illustrative example.

Q3 (1/10). Provide a parallel implementation (OpenMP) of the two versions (mm.c and mm2.c) and provide the results obtained from executing them using from one (1) to four (4) threads. You are requested to follow methodology used in the first assignment (i.e., automating multiple executions providing statistical results).

6. Profiling Applications

In order to study the behavior of OpenMP 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 directory:

```
/share/apps/papi/
```

In the example `flops.c` and `flops2.c` provided with this assignment, an example of PAPI usage is shown for the codes used in the previous section. 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/share/apps/papi/include/ flops.c /share/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.

Q4 (0.5/10). Why do you think there is a 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/share/apps/papi/include/  counters.c  /share/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 assignment 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 assignment, 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:

```
/share/apps/papi/bin/papi_avail
```

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

```
[@eimtarqso]$ /share/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
```

Q5 (0.5/10). What hardware counters would you use to study the differences between the two implementations? Why?

Q6 (2/10). Provide the code where you use hardware counters to quantify the differences between the two examples. Provide the results obtained.

7. OpenMP performance evaluation

In this section of the assignment, we will study how the OpenMP scheduling policies impact load imbalance and, as a result, application performance. We will "hack" and evaluate the NASA Parallel Benchmarks (NPB) – please see https://www.nas.nasa.gov/publications/npb.html.

These benchmarks include five kernels and three pseudo applications that are very useful for conducting performance evaluation studies. These benchmarks are organized in classes, which defines the problem size. In this section, we will focus on classes W, A, and B. Please note that the large test problems (i.e., classes D, E, F) may impact the cluster's use due to very long executions, so please refrain from using them.

A performance study has to be performed with the combination of the following parameters:

- Benchmark – kernel FT and pseudo applications SP and LU
- Size of the problem W, and A
- Number of threads (1, 2, 3, 4) – i.e., the number of threads available the nodes

The binaries of the benchmarks can be found in the following path:

```
/share/apps/aca/benchmarks/NPB3.2/NPB3.2-OMP/bin
```

Another copy of the benchmarks is also available in `/tmp/cat2/NPB3.2-OMP/`

Note: Please copy /tmp/cat2 locally in a directory under your $HOME.

Please note that the output of the benchmarks' execution provides the execution time. Please see below the example for SP class A (`bin/sp.A`) using 4 threads:

```
 NAS Parallel Benchmarks (NPB3.2-OMP) - SP Benchmark

 No input file inputsp.data. Using compiled defaults
 Size:  64x 64x 64
 Iterations: 400    dt:    0.001500
 Number of available threads:     4

 Time step    1
 Time step   20
 Time step   40
(...)
 Time step  380
 Time step  400
 Verification being performed for class A
 accuracy setting for epsilon =  0.1000000000000E-07
 Comparison of RMS-norms of residual
           1 0.2479982239930E+01 0.2479982239930E+01 0.3975343169776E-13
           2 0.1127633796437E+01 0.1127633796437E+01 0.9786525465665E-13
           3 0.1502897788877E+01 0.1502897788877E+01 0.4181164125655E-13
           4 0.1421781621170E+01 0.1421781621170E+01 0.7902378847720E-13
           5 0.2129211303514E+01 0.2129211303514E+01 0.4004540467660E-13
 Comparison of RMS-norms of solution error
           1 0.1090014029782E-03 0.1090014029782E-03 0.4214906029077E-12
           2 0.3734395176929E-04 0.3734395176928E-04 0.1235711615405E-12
           3 0.5009278540654E-04 0.5009278540654E-04 0.1745037722443E-13
           4 0.4767109393954E-04 0.4767109393953E-04 0.2334040164727E-12
           5 0.1362161339921E-03 0.1362161339921E-03 0.5432307914157E-13
 Verification Successful


 SP Benchmark Completed.
 Class           =                        A
 Size            =             64x  64x  64
 Iterations      =                      400
 Time in seconds =                    29.01
 Total threads   =                        4
 Avail threads   =                        4
 Mop/s total     =                  2930.80
 Mop/s/thread    =                   732.70
 Operation type  =           floating point
 Verification    =               SUCCESSFUL
 Version         =                      3.2
 Compile date    =              04 Oct 2013

 Compile options:
    F77          = gfortran
    FLINK        = $(F77)
    F_LIB        = (none)
    F_INC        = (none)
    FFLAGS       = -O  -fopenmp -mcmodel=medium
    FLINKFLAGS   = -O  -fopenmp -mcmodel=medium
    RAND         = (none)


 Please send all errors/feedbacks to:

 NPB Development Team
 npb@nas.nasa.gov
```

Q7 (2/10). Provide the requested performance study in terms of execution time and speedup.

8. OpenMP scheduling policies and load imbalance

In this section of the assignment, we will study how the OpenMP scheduling policies impact load imbalance and, as a result, application performance. We will "hack" and evaluate the NASA Parallel Benchmarks (NPB) used in the previous section to modify the OpenMP scheduling policy used by these benchmarks.

A performance study has to be performed with the combination of the following parameters:

- Benchmark –pseudo applications SP, and LU.
- Size of the problem W.
- Number of threads = 4
- Scheduling policies: (static), (dynamic,k), and (guided,k) with:
  - k=1
  - k>1 (at least one value of your choice)

Some sample instructions to modify the scheduling policy in the selected benchmarks and compile them are provided below. Please note that although the benchmarks are implemented in Fortran, the OpenMP pragmas have a similar form.

Copy the NPB into a directory under your $HOME:

```
cd
cp -r /share/apps/aca/benchmarks/NPB3.2/NPB3.2-OMP .
cd NPB3.2-OMP
```

#check the scheduling policy:

```
grep schedule SP/*.f
```

```
...
SP/rhs.f:!$omp do schedule(static)
...
```

#compile and copy file:

```
make sp CLASS=W
cp bin/sp.W bin/sp.W.static
```

#modify the scheduling policy:

```
sed -i 's/schedule(static)/schedule(dynamic,1)/g' SP/*.f
```

#check again the scheduling policy:

```
grep schedule SP/*.f
```

```
...
SP/rhs.f:!$omp do schedule(dynamic,1)
...
```

#compile and copy file:

```
make sp CLASS=W
cp bin/sp.W bin/sp.W.dynamic.1
```

#repeat the steps for guided and other configurations

In your SGE scripts you can use `bin/sp.W.static`, `bin/sp.W.dynamic.1`, etc.

Although it is not mandatory (but strongly encouraged), this process can be automated.

For this part of the assignment we will also use Extrae (https://tools.bsc.es/extrae), which is part of a performance evaluation toolkit developed by the Barcelona Supercomputing Center.

In order to use extrae you must define the following variables and execute the following:

```
export EXTRAE_HOME=/share/apps/extrae
cp /share/apps/extrae/share/example/OMP/extrae.xml YOUR_DIRECTORY
source /share/apps/extrae/etc/extrae.sh
export EXTRAE_CONFIG_FILE=YOUR_DIRECTORY/extrae.xml
export LD_PRELOAD=/share/apps/extrae/lib/libomptrace.so
```

then, you can execute your programs as usually, for example (for SP class W, static):

```
./sp.W.static
```

As a result of the execution of the instrumented application you will obtain a trace which is composed of three files: `sp.W.static.prv`, `sp.W.static.row` and `sp.W.static.pcf`.

In order to visualize and analyze the obtained trace you will use Paraver (https://tools.bsc.es/paraver). This tool is available for multiple platforms and you will need to install it in your system.

Q8 (2/10). Provide the requested performance study in terms of execution time and speedup. Elaborate the results in terms of load imbalance (qualitatively) based on your findings using Extrae/Paraver.

9. Parallelization problem with OpenMP

A synthetic sequential program is provided in `q9.c`.

Q9 (1/10). Provide a parallel implementation (OpenMP) of the sequential program in q9.c. You can make code transformations as long as they do not modify the program functionality. Provide your design decisions to improve performance/balance and performance evaluation.

Submission format and due date

Please submit your assignment as a single TAR or ZIP file, containing:

- A single PDF file with the responses to the questions, including the scripts, graphs, etc.
- A directory named "files" with all the relevant materials (scripts, source code, Gnuplot scripts, latex document, latex report, etc.)

The TAR or ZIP file must be named "LastnamesFirstnameCAT2.tar" (or .zip).

Brevity and concretion are a plus.

IMPORTANT: Submissions in other formats or organizations will be returned without grading.

Assignments are due: 11 April 2023.