PEC 2

Pablo Riutort Grande

April 16, 2023

M0.538 - HIGH PERFORMANCE COMPUTING

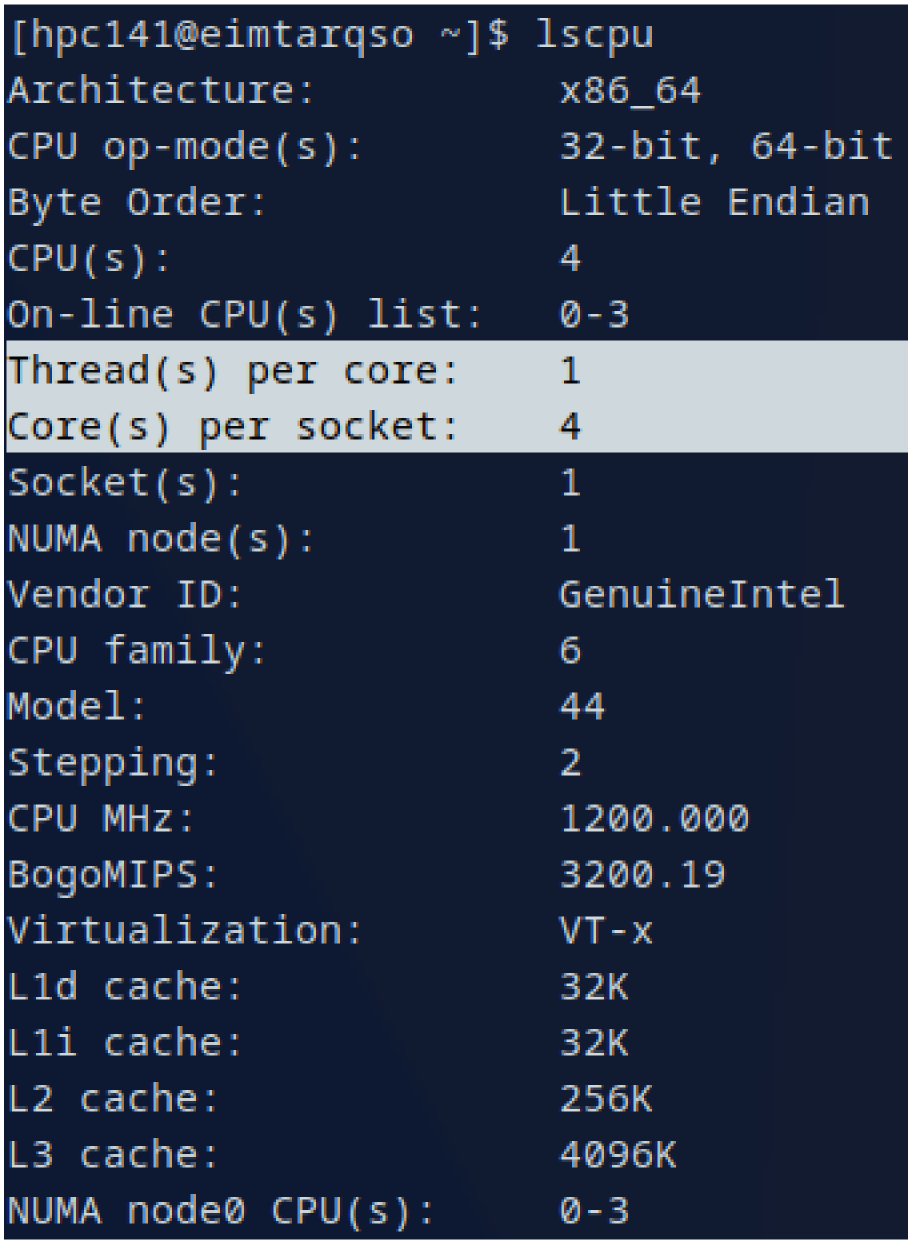
MU Ingeniería Informática / MU Ingeniería Computacional y Matemática

Estudios de Informática, Multimedia y Telecomunicación

1 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.

In general and by default OpenMP uses: OpenMP threads = CPU cores. A user can use as many OpenMP threads as there are available cores on the node(s) you are using. We can get this information from the cluster by executing:

```
lscpu
```



```
[hpc141@eimtarqso ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 44
Stepping:              2
CPU MHz:               1200.000
BogoMIPS:              3200.19
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              4096K
NUMA node0 CPU(s):     0-3
```

Figure 1: lscpu

So, in our cluster we can use up to 4 OpenMP threads.

The number of OpenMP threads a user can use in a cluster depends on several factors, including the number of cores and memory available on the nodes, the specific configuration of the cluster and the workload of other users on the system.

Inconvenients of using a different configuration for the threads in a cluster for OpenMP:

- Scheduling: Trying to use more threads than allowed by the job scheduler may lead in a job to be queued or rejected.
- Performance: Using a different number of threads than the number of available cores can lead overutilization or underutilization of system resources, which can cause performance issues and may even lead to system instability.
- Memory usage: Each OpenMP thread requires some amount of memory to operate, so using too many threads can lead to excessive memory usage.

2 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.

```
 #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];
    }
```

We can make use of the sum clause suming all elements in a like this:

```
  // Calculate the sum of elements stored in the vector "a" in parallel
  #pragma omp parallel for reduction(+:sum)
  for(i=0; i<N; i++){
    sum += a[i];
  }

```

**Supposing 'a' has been initialized.*

3 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).

Parallel implementations:

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <memory.h>
#include <malloc.h>
#include <omp.h>

#define SIZE 1000

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

  float matrixa[SIZE][SIZE], matrixb[SIZE][SIZE], mresult[SIZE][SIZE];
  int i,j,k;

  /* Initialize the Matrix arrays */
  #pragma omp parallel for private(i, j)
  for (i = 0; i < SIZE; i++) {
    for (j = 0; j < SIZE; j++) {
      mresult[i][j] = 0.0;
      matrixa[i][j] = matrixb[i][j] = rand()*(float)1.1;
    }
  }

  /* Matrix-Matrix multiply */
  #pragma omp parallel for private(i, j, k) shared(matrixa, matrixb, mresult)
  for (i = 0; i < SIZE; i++) {
    for (j = 0; j < SIZE; j++) {
      for (k = 0; k < SIZE; k++) {
        mresult[i][j] += matrixa[i][k] * matrixb[k][j];
      }
    }
  }

  exit(0);
}
```

Listing 1: pmm.c: Parametrization of mm.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <memory.h>
#include <malloc.h>
#include <omp.h>

```

```c
#define SIZE 1000

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

  float matrixa[SIZE][SIZE], matrixb[SIZE][SIZE], mresult[SIZE][SIZE];
  int i,j,k;

  /* Initialize the Matrix arrays */
  #pragma omp parallel for collapse(2)
  for ( i=0; i<SIZE; i++ ){
    for ( j=0; j<SIZE; j++ ){
      mresult[i][j] = 0.0;
      matrixa[i][j] = matrixb[i][j] = rand()*(float)1.1;
    }
  }

  /* Matrix-Matrix multiply */
  #pragma omp parallel for private(k, i, j)
  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];
    }
   }
  }

  exit(0);
}
```

Listing 2: pmm2.c: Parametrization of mm2.c

Alongside, we developed the next scripts:

```bash
#!/bin/bash
#$ -cwd
#$ -S /bin/bash
#$ -N pmmVERSION_THREADS
#$ -o pmmVERSION_THREADS.out.$JOB_ID
#$ -e pmmVERSION_THREADS.out.$JOB_ID
#$ -pe openmp THREADS

time ./pmmVERSION
```

Listing 3: pmm.sge: SGE template file

```sh
#!/bin/sh

for v in {1,2}; do
for i in {1,2,3,4}; do
  export OMP_NUM_THREADS=$i;
  cp pmm.sge pmm$i.sge;
  sed -i 's/THREADS/'$i'/g' pmm$i.sge
  sed -i 's/VERSION/'$v'/g' pmm$i.sge
  qsub pmm$i.sge
done;
done;
```

Listing 4: run.sh: Helper script

Each parametrization of mm is called pmm1 or pmm2, depending on the version of mm. pmm.sge contains two sentinel words to change with sed: VERSION and THREAD. VERSION refers to either 1 or 2 in pmm1 and pmm2. THREADS is the number of threads OpenMP will use, from 1 to 4. run.sh is a script that will generate each SGE iterating over versions and number of threads. For each version of the file will export the number of threads, will make changes in pmm.sge file and send the file to the queue. After some scripting we can do some statistical results with Python:

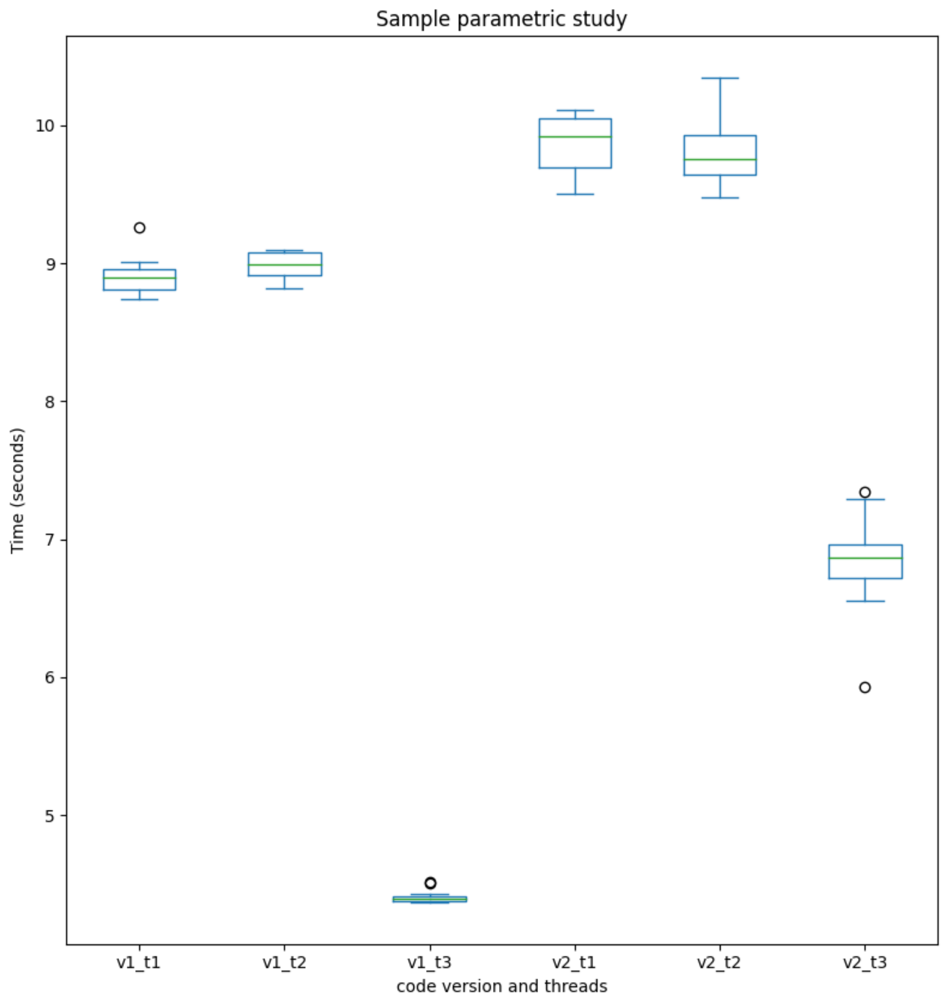


Figure 2: Statistic analysis

```python
import pandas as pd

from matplotlib import pyplot as plt
from os import walk

data = {}
# get data from results files
```

```
for dirpath, dirnames, filenames in walk("."):
    for filename in filenames:
        if not filename.endswith(".py"):
            fname, threads = filename.split(".")[0].split("_")
            # values with 4 threads are discarded because they do not provide enough values
            if threads == "4":
                continue
            version = fname.split("pmm")[1]
            key = f"v{version}_t{threads}"
            if not key in data:
                data[key] = []
            with open(filename, "r") as _file:
                content = _file.readlines()
                time = content[1].split("\t")[1].strip().split("m")[1].rstrip("s")
                data[key].append(float(time))

df = pd.DataFrame(data)
df = df.reindex(sorted(data.keys()), axis=1)

p = df.plot.box()
p.set_title("Sample parametric study")
p.set_xlabel("code version and threads")
p.set_ylabel("Time (seconds)")
plt.show()
```

4 Why do you think there is a difference in execution time and MFLOPs?

```
    [hpc141@eimtarqso ~]$ ./flops; ./flops2
Real_time:  11.539018
Proc_time:  11.511601
Total flpins: 2000964624
MFLOPS:   173.821579
flops.c PASSED
Real_time:  13.255196
Proc_time:  13.223212
Total flpins: 2003602539
MFLOPS:   151.521622
flops2.c  PASSED
```

Listing 5: flops execution

The difference between process time and MFLOPs is due to the fact that they are measuring different aspects of the program's performance.

Process time is the amount of time that the CPU spends executing the program. MFLOPs, on the other hand, is a measure of the program's floating-point operations per second (FLOPs) normalized to millions of FLOPs per second (MFLOPs). It gives an indication of the speed of the program's arithmetic operations.

The process time measures the total time taken by the program to execute, including any overhead from the operating system. Meanwhile, MFLOPs measures the number of floating-point operations performed per second, so it reflects the efficiency of the algorithm used in the program.

5 What hardware counters would you use to study the differences between the two implementations? Why?

IPC measures the number of instructions that can be executed per cycle, indicating how well a CPU can utilize its resources to execute instructions. High IPC values suggest that a CPU is executing instructions efficiently, whereas low IPC values may indicate that there are instruction dependencies, pipeline stalls, or cache misses, all of which can negatively impact performance.

Cache misses are a common cause of reduced IPC and code inefficiency. When the CPU tries to access data that is not in the cache, a cache miss occurs. To improve code efficiency, it is important to minimize the number of cache misses.

With PIPA hardware counters we can identify areas of code that may be causing cache misses and reducing IPC. By analyzing hardware performance counters related to cache behavior, PIPA can identify the specific types of cache misses that are occurring, such as L1 or L2 cache misses. This information can then be used to optimize the code by reducing the number of cache misses and improving IPC.

In our case, L2 cache hardware counters are not available, we make use of the next counters:

- $PAPI\_TOT\_INS$: Instructions completed
- $PAPI\_FP\_OPS$: Floating point operations
- $PAPI\_L1\_DCM$: Level 1 data cache misses
- $PAPI\_L1\_ICM$: Level 1 instruction cache misses

Level 1 (L1) cache misses are the misses that occur in the primary cache, typically the smallest and fastest cache in the system. These misses are often the fastest to resolve since they are the first level of cache accessed by the processor.

6 Provide the code where you use hardware counters to quantify the differences between the two examples. Provide the results obtained.

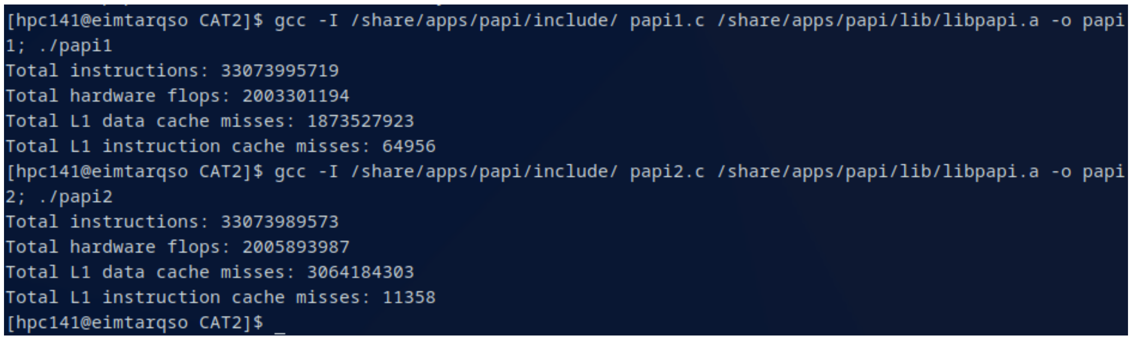
With the provided examples of matrix multiplication we can make a study on how changing the loops may impact its overall performance, hereby is the code provided adding those PIPA counters:

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <memory.h>
#include <malloc.h>
#include <papi.h>

#define SIZE 1000
#define NUM_EVENTS 4

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

  float matrixa[SIZE][SIZE], matrixb[SIZE][SIZE], mresult[SIZE][SIZE];
  int i,j,k;
  int events[NUM_EVENTS] = {PAPI_TOT_INS, PAPI_FP_OPS, PAPI_L1_DCM, PAPI_L1_ICM },
     ret;
  long long values[NUM_EVENTS];

  if (PAPI_num_counters() < NUM_EVENTS) {
    fprintf(stderr, "No hardware counters here, or PAPI not supported.\n");
    exit(1);
  }

  if ((ret = PAPI_start_counters(events, NUM_EVENTS)) != PAPI_OK) {
    fprintf(stderr, "PAPI failed to start counters: %s\n", PAPI_strerror(ret));
    exit(1);
  }

  /* Initialize the Matrix arrays */
  for ( i=0; i<SIZE*SIZE; i++ ){
    mresult[0][i] = 0.0;
    matrixa[0][i] = matrixb[0][i] = rand()*(float)1.1; }

  /* Matrix-Matrix multiply */
  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];

  if ((ret = PAPI_read_counters(values, NUM_EVENTS)) != PAPI_OK) {
    fprintf(stderr, "PAPI failed to read counters: %s\n", PAPI_strerror(ret));
    exit(1);
  }

  printf("Total instructions: %lld\n", values[0]);
  printf("Total hardware flops: %lld\n",values[1]);
  printf("Total L1 data cache misses: %lld\n", values[2]);
  printf("Total L1 instruction cache misses: %lld\n", values[3]);

  exit(0);
}
```

Listing 6: papi1.c from counters.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <memory.h>
#include <malloc.h>
#include <papi.h>

#define SIZE 1000
#define NUM_EVENTS 4

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

  float matrixa[SIZE][SIZE], matrixb[SIZE][SIZE], mresult[SIZE][SIZE];
  int i,j,k;
  int events[NUM_EVENTS] = {PAPI_TOT_INS, PAPI_FP_OPS, PAPI_L1_DCM, PAPI_L1_ICM},
    ret;
  long long values[NUM_EVENTS];

  if (PAPI_num_counters() < NUM_EVENTS) {
    fprintf(stderr, "No hardware counters here, or PAPI not supported.\n");
    exit(1);
  }

  if ((ret = PAPI_start_counters(events, NUM_EVENTS)) != PAPI_OK) {
    fprintf(stderr, "PAPI failed to start counters: %s\n", PAPI_strerror(ret));
    exit(1);
  }

  /* Initialize the Matrix arrays */
  for ( i=0; i<SIZE*SIZE; i++ ){
    mresult[0][i] = 0.0;
    matrixa[0][i] = matrixb[0][i] = rand()*(float)1.1; }

  /* Matrix-Matrix multiply */
  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];

  if ((ret = PAPI_read_counters(values, NUM_EVENTS)) != PAPI_OK) {
    fprintf(stderr, "PAPI failed to read counters: %s\n", PAPI_strerror(ret));
    exit(1);
  }

  printf("Total instructions: %lld\n", values[0]);
  printf("Total hardware flops: %lld\n",values[1]);
  printf("Total L1 data cache misses: %lld\n", values[2]);
  printf("Total L1 instruction cache misses: %lld\n", values[3]);

  exit(0);
}
```

Listing 7: papi2.c from counters2.c



```
[hpc141@eimtarqso CAT2]$ gcc -I /share/apps/papi/include/ papi1.c /share/apps/papi/lib/libpapi.a -o papi
1; ./papi1
Total instructions: 33073995719
Total hardware flops: 2003301194
Total L1 data cache misses: 1873527923
Total L1 instruction cache misses: 64956
[hpc141@eimtarqso CAT2]$ gcc -I /share/apps/papi/include/ papi2.c /share/apps/papi/lib/libpapi.a -o papi
2; ./papi2
Total instructions: 33073989573
Total hardware flops: 2005893987
Total L1 data cache misses: 3064184303
Total L1 instruction cache misses: 11358
[hpc141@eimtarqso CAT2]$ _
```

Figure 3: PAPI analysis

papi2 made 6140 less total instructions but 2592793 more hardware flops than papi1. papi2 has over 1190656380 data cache misses than papi1 but has 53598 instruction cache misses less than papi1.

	papi1	papi2
Total Instructions	33073995719	33073989573
Total hardware flops	2003301194	2005893987
L1 data cache misses	1873527923	3064184303
L1 instruction cache misses	64956	11358

Table 1: papi1 and papi2 executions

7 Provide the requested performance study in terms of execution time and speedup.

We developed two scripts for this exercise. One that deals with the execution of the SGE file and a SGE file template. The first script will be run 10 times to gather the results of the benchmarks.

```
#!/bin/bash
#$ -cwd
#$ -S /bin/bash
#$ -N nasa_BENCHMARK_SIZE_THREADS
#$ -o nasa_BENCHMARK_SIZE_THREADS.out.$JOB_ID
#$ -e nasa_BENCHMARK_SIZE_THREADS.out.$JOB_ID
#$ -pe openmp THREADS

/home/hpc141/CAT2/NPB3.2-OMP/bin/BENCHMARK.SIZE
```

Listing 8: nasa.sge: NPB SGE template file

```
#!/bin/sh

for benchmark in {ft,sp,lu}; do
for size in {w,a}; do
for threads in {1,2,3,4}; do
  export OMP_NUM_THREADS=$i;
  sge=nasa$benchmark_$size_$threads.sge;
  cp nasa.sge $sge
  sed -i 's/THREADS/'$threads'/g' $sge
  sed -i 's/BENCHMARK/'$benchmark'/g' $sge
  sed -i 's/SIZE/'$size'/g' $sge
  qsub $sge
done;
done;
done;
```

Listing 9: nasa.sh: Helper script

nasa.sh is a script that will generate each SGE iterating over benchmark, sizes and number of threads.

After 10 executions we have enough to make a study in terms of execution and speedup with Python:

```
import pandas as pd

from matplotlib import pyplot as plt
from os import walk

data = {}
# get data from results files
```

```
for _, dirnames, _ in walk("."):
    for _dir in dirnames:
        for _, _, filenames in walk(_dir):
            for filename in filenames:
                if not filename.endswith(".sge"):
                    _, benchmark, size, trail = filename.split("_")
                    threads = trail.split(".")[0]
                    key = f"{benchmark}.{size}-{threads}"
                    if not key in data:
                        data[key] = []
                    with open(f"{_dir}/{filename}", "r") as _file:
                        content = _file.readlines()
                    # find our time in seconds
                    t_in_seconds_str = [t for t in content if t.strip().startswith
    ("Time in seconds")]
                    t_in_seconds = float(t_in_seconds_str[0].strip().split("=")
    [-1])
                    data[key].append(float(t_in_seconds))

df = pd.DataFrame(data)
df = df.reindex(sorted(data.keys()), axis=1)

p = df.plot.box()
p.set_title("Performance study")
p.set_xlabel("Benchmark version and threads")
p.set_ylabel("Time (seconds)")
plt.show()
```

Listing 10: Python helper script

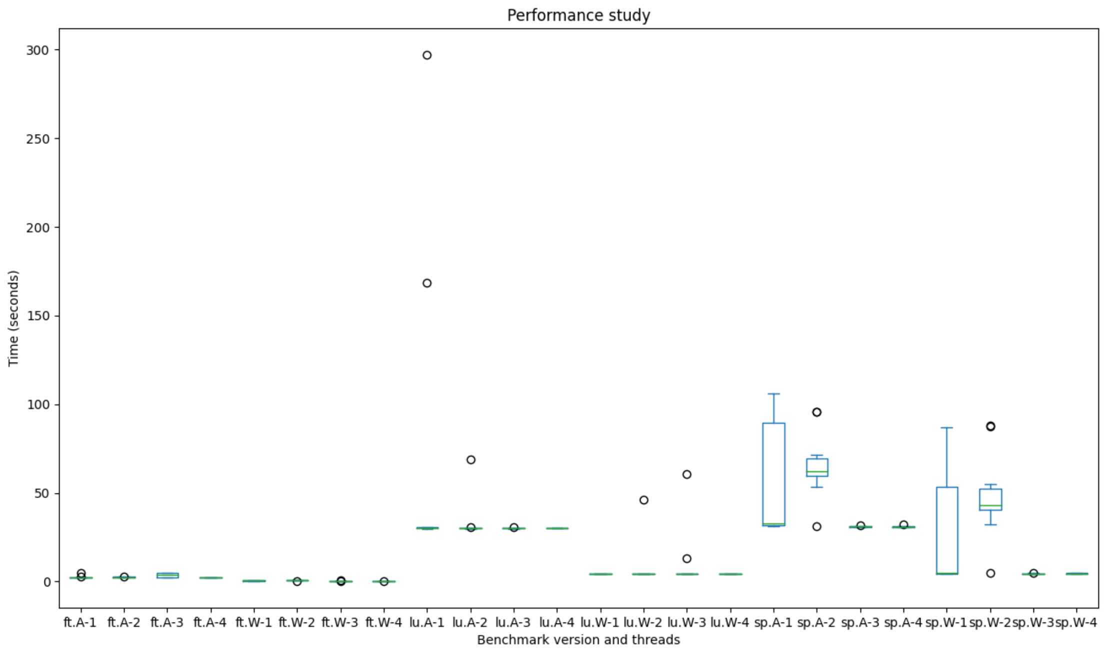


Figure 4: Performance study

8 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.

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

#define N 16000
#define T 4

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

  long long A[T][N], B[T][N], avg[T];
  int i,j,k,l,sum,elems;

  srand(time(NULL));

  #pragma omp parallel for private(j)
  for(i=0; i<T; i++){
   for(j=0; j<N; j++){
    if(i<T/2){
     A[i][j]=N/2+rand()%(N/2);
    }
    else{
     A[i][j]=rand()%(N/8);
    }
   }
  }

  #pragma omp parallel for private(j,k,l,sum)
  for(i=0; i<T; i++){
   for(j=0; j<N; j++){
    sum=0;
    for(k=0; k<A[i][j]; k++){
     for(l=0; l<T; l++){
      if(l!=i){
       sum=(sum+A[l][j])%(N/8);
      }
     }
    }
    B[i][j]=(A[i][j]+sum)%N;
   }
  }

  #pragma omp parallel for private(j,k,l,sum) shared(avg)
  for(i=0; i<2; i++){
   elems=N/2+rand()%(N/4);
   sum=0;
   for(j=0; j<N; j++){
     for(k=0; k<A[i][j]; k++){
       for(l=0; l<T; l++){
         sum=(sum+A[l][j])%(N/4);
```

```
        }
      }
    }
    avg[i]=sum/elems;
  }

  #pragma omp parallel for private(j)
  for(i=0; i<T; i++){
    for(j=0; j<N; j++){
      B[i][j]+=A[i][j]+avg[1];
    }
  }

  exit(0);
}
```

We use the "private" clause to ensure that each thread has its own copy of the loop index variables and shared clauses to ensure that the threads do not interfere with each other's data. We have also added OpenMP directives to parallelize the outer loops.

[Knuth(1986)]

References

[Knuth(1986)] D. E. Knuth. *The TEX Book*. Addison-Wesley Professional, 1986.