Supercomputers Architecture

Hands on 4

November 24th, 2015

Constantino Gomez

Albert Segura

Cristobal Ortega

Exercise 1:

a) Compile and execute the test.c program. Use the flag "-O0".

```
$ gcc test.c -o test_O0 -O0
test.c:19: warning: return type defaults to 'int'
test.c: In function 'main':
test.c:31: warning: implicit declaration of function 'print_times'
test.c: At top level:
test.c:35: warning: return type defaults to 'int'
```

b) Create a new program test.ji.c based in .test.c changing "m1[j][i] = 8.0/m2[j][i];" by " m1[i][j] = 8.0/m2[i][j];

c) Justify why the execution time of the two loops are so different.

```
$ /usr/bin/time -f %e ./test_O0
 25079.92 mSec
25.45
$ /usr/bin/time -f %e ./test_ij_O0
 4601.81 mSec
4.94
$
```

The difference comes from the change we made (obviously), in the original version of the program the access to the matrix is done like this:

```
for(i = 0; i < SIZE; ++i)
      for(j = 0; j < SIZE; ++j)
            m1[j][i] …
```

This means we are changing of row in each iteration (second loop), and since the SIZE is 8000 every row of the matrix have length of 8000*sizeof(double), this length is bigger than the length of a cache line and since the matrix is SIZE*SIZE, it does not fit in cache, as a result, in every iteration the program is missing in cache and has to go to main memory to get the value of m1[j][i].

The second version of the program is accesing by column, that is a normal pattern that a data prefetcher can detect and anticipate its behavior.

Exercise 2: Execute the previous example test.c program using different flags of Intel compiler

```
$ icc -O0 test.c -o test
$ echo "icc -O0"
icc -O0
$ ./test
 27510.94 mSec
$ icc -O2 test.c -o test
$ echo "icc -O2"
icc -O2
$ ./test
 0.00 mSec
$ icc -O3 test.c -o test
$ echo "icc -O3"
icc -O3
$ ./test
 0.00 mSec
```

Exercise 3: using the man icc explain why the execution time is different between them. Tip: search what is “dead code elimination (DCE)”.

O1:
this option:

- Enables global optimization; this includes data-flow analysis, code motion, strength reduction and test replacement, split-lifetime analysis, and instruction scheduling.
- Disables inlining of some intrinsics.

O2:

- Includes O1
- Inlining of intrinsics
- Intra-file interprocedural optimization
- The following capabilities for performance gain:
    - …
    - dead-code elimination
    - …

O3:

- Includes O2

The program is not using the m1 matrix, as input of anything, so looks like the compiling is just erasing code not used like the initialization of the matrix and the foo() calls.
That’s why we are getting a 0 seconds execution.

Exercise 4: Do the same with gcc compiler. What is your opinion?

```
$ gcc -O0 test.c -o test -w
$ ./test
 26460.83 mSec
$ gcc -O2 test.c -o test -w
$ ./test
 21243.12 mSec
$ gcc -O3 test.c -o test -w
$ ./test
 11366.70 mSec
```

GCC is not eliminating code, this can be useful if the programmer wants to check something as we are doing now (for timing of some function), using gcc he can timing normally, using ICC the programmer has to be careful to use the output produced (at least printing it) so the compiler does not erase that.
But for releasing a software ICC can be helpful for reducing the size of the binary generated.

Exercise 5: Check the current module. Run the same program (e.g. your matrix multiplication program from exercise 10 in hands-on 3 for N=16384 and 64 processors) in two different environments: OpenMPI and IntelMPI. Compare the execution time obtained in boths cases and give your opinion. Include in your answer the data obtained and the JSF files used.

```
$ module list
Currently Loaded Modulefiles:
  1) intel/13.0.1     2) openmpi/1.8.1   3) transfer/1.0    4)
bsc/current
```

	2 threads	**4 threads**	**8 threads**	**16 threads**
OpenMPI	1,195	1,12	1,44	2,09
IntelMPI	1,75	1,695	2,175	2,955

Data collected from executions of trapezoidal with the same input as used in the hands-on 3

LSF:

```
$ cat exercise_5.sh
#BSUB -J trapezoidal_omp
#BSUB -o output_%J.out
#BSUB -e output_%J.err
#BSUB -W 02:00
#BSUB -n 16
#BSUB -R "span[ptile=16]"
#BSUB -x

#Executing

IMPI="impi"
OMPI="openmpi"

module unload openmpi
module unload impi


module load $IMPI

for i in 2 4 8 16
do
/usr/bin/time -f %e -ao ${i}_threads_$IMPI.txt mpirun -np $i
./trapezoidal_mpi < input.txt
```

```
/usr/bin/time -f %e -ao ${i}_threads_$IMPI.txt mpirun -np $i
./trapezoidal_mpi < input.txt
done

module unload $IMPI
module load $OMPI

for i in 2 4 8 16
do
/usr/bin/time -f %e -ao ${i}_threads_$OMPI.txt mpirun -np $i
./trapezoidal_mpi < input.txt
/usr/bin/time -f %e -ao ${i}_threads_$OMPI.txt mpirun -np $i
./trapezoidal_mpi < input.txt
done
```

Exercise 6:
a) Create the bash script file top.sh with the following code:.
The idea is to launch in background the top.sh command and then execute the standard mpirun command with any of the MPI jobs previously executed during this training.
b) Create a JSF or take one used in a previous exercise and add the following line before executing the mpirun: ./top.sh &

```
$ cat exercise_6.sh
#BSUB -J trapezoidal_omp
#BSUB -o output_%J.out
#BSUB -e output_%J.err
#BSUB -W 02:00
#BSUB -n 16
#BSUB -R "span[ptile=16]"
#BSUB -x

#Executing

./top.sh &
/usr/bin/time -f %e  mpirun -np $i ./trapezoidal_mpi < input.txt
```

c) Submit the JSF again Check the .out file. What is its content? Please, explain

```
top - 14:46:41 up 25 days, 14 min,  0 users,  load average: 3.45, 3.75, 3.18
Tasks: 225 total,  17 running, 208 sleeping,   0 stopped,   0 zombie
Cpu(s): 65.9%us,  1.5%sy,  0.0%ni, 32.6%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   129172M total,        6718M used,   122454M free,          0M buffers
Swap:  11442M total,         0M used,       11442M free,   2483M cached

  PID USER     PR  NI  VIRT  RES  SHR S   %CPU %MEM TIME+  COMMAND
```

```
31488 sam14021  20   0  429m  44m  38m R    20  0.0   0:00.42 trapezoidal_mpi
31481 sam14021  20   0  429m  44m  38m R    18  0.0   0:00.40 trapezoidal_mpi
31482 sam14021  20   0  429m  44m  38m R    18  0.0   0:00.40 trapezoidal_mpi
31496 sam14021  20   0  429m  44m  38m R    18  0.0   0:00.39 trapezoidal_mpi

…
```

We can see that we are getting output from the script top.sh that we executed in background with &, then we can see the output from our program normally, if the program prints in the middle of the execution we would see the output from our top.sh and our program mixed, it would be better redirect top.sh to some file.

Exercise 7 (optional):
a) Create the bash script file vmstat.sh with the following code:.
b) Create a LSF job using the vmstat.sh bash script. The idea is to launch in background the vmstat.sh command and then execute the standard mpirun command with any of the MPI jobs previously executed during this training.

```
$ cat exercise_7.sh
#BSUB -J vmstat
#BSUB -o output_%J.out
#BSUB -e output_%J.err
#BSUB -W 02:00
#BSUB -n 16
#BSUB -R "span[ptile=16]"
#BSUB -x

#Executing

./vmstat.sh &
/usr/bin/time -f %e  mpirun -np 16 ./trapezoidal_mpi < input.txt
```

c) Check the meaning of all the fields shown by the vmstat command

```
$ cat output_1306565.out
procs -----------memory---------- ---swap-- -----io---- -system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so bi    bo   in   cs us sy id wa st
 1  0   0 126497220    0 2421780      0       0       0       11      0       1 64  2 34  0  0
With n = 1024 trapezoids, our estimate
of the area from 0.000000 to 3.000000 = 9.000004291534424
 0  0   0 126512732    0 2422672      0       0       0       1 12811 30418  9  2 89  0  0
```

If we look up what vmstat does:

```
$ man vmstat
```

vmstat - Report virtual memory statistics

And the values of each field are also described:
FIELD DESCRIPTION FOR VM MODE

Procs

r: The number of processes waiting for run time.
b: The number of processes in uninterruptible sleep.

Memory

swpd: the amount of virtual memory used.
free: the amount of idle memory.
buff: the amount of memory used as buffers.
cache: the amount of memory used as cache.
inact: the amount of inactive memory. (-a option)
active: the amount of active memory. (-a option)

Swap

si: Amount of memory swapped in from disk (/s).
so: Amount of memory swapped to disk (/s).

IO

bi: Blocks received from a block device (blocks/s).
bo: Blocks sent to a block device (blocks/s).

System

in: The number of interrupts per second, including the clock.
cs: The number of context switches per second.

CPU

These are percentages of total CPU time.
us: Time spent running non-kernel code. (user time, including nice time)
sy: Time spent running kernel code. (system time)
id: Time spent idle. Prior to Linux 2.5.41, this includes IO-wait time.
wa: Time spent waiting for IO. Prior to Linux 2.5.41, included in idle.
st: Time stolen from a virtual machine. Prior to Linux 2.6.11, unknown.

And in our vmstat.sh script we do `vmstat 3 3`, this means 3 seconds of delay between each reading of the stat of the memory, and 3 total reads.
But we just see 2 because our program takes less time than that time (also in vmstat.sh we have a sleep of 2 seconds)

Exercise 8 (OPTIONAL):

a) Try the following command.
Check and explain the output generated by the PERF performance tool ,
Help about PERF command can be found at the following page :
https://perf.wiki.kernel.org/index.php/Main_Page
b) Create the bash script file perf.sh with the following code:.
c) Create a LSF job using the previous perf.sh bash script. The idea is to launch it in background and then execute the standard mpirun command with any of the MPI jobs previously executed during this training.

```
$ cat exercise_8.sh
#BSUB -J perf
#BSUB -o output_%J.out
#BSUB -e output_%J.err
#BSUB -W 02:00
#BSUB -n 16
#BSUB -R "span[ptile=16]"
#BSUB -x

#Executing

./perf.sh mpirun -np 16 ./trapezoidal_mpi < input.txt
```

and the output produced by perf is:

```
$ cat mpirun.out.perf

 Performance counter stats for 'mpirun -np 16 ./trapezoidal_mpi':

       6059.869121 task-clock              #      1.919 CPUs utilized
       180,942 context-switches            #      0.030 M/sec
              12,075 CPU-migrations        #      0.002 M/sec
              79,103 page-faults           #      0.013 M/sec
       16,016,710,603 cycles                      #      2.643 GHz
       6,961,270,175 stalled-cycles-frontend   #   43.46% frontend cycles idle
       4,807,376,420 stalled-cycles-backend#   30.01% backend  cycles idle
       21,030,532,071 instructions                #      1.31  insns per cycle
                                          #      0.33  stalled cycles per insn
       5,541,427,818 branches             #  914.447 M/sec
       13,571,929 branch-misses           #      0.24% of all branches

       3.157099393 seconds time elapsed
```

Perf shows software and hardware information like context-switches, cycles, instructions, etc. but adding the right flags to the perf execution we could read any hardware performance counter from the processor.

Exercise 9: Launch some of the previous codes used in this course (e.g. your matrix multiplication program from exercise 10 in hands-on 3 for N=32768 and 16 processors). Inspect the log file generated. Determine and describe the evolution of memory and CPU use. Discuss your conclusions with the teacher at lab session.

In the profile we can see in top our top.sh, vmstat, and our real command (also bash and that kind of things…). We see that when our program goes to execution the CPU usage increases (top reports such thing), also when our program is no running in the CPU we have swapped memory. Our program executing is trapezoidal's rule so the execution time is short compared to so much sleep do it between the execution and profiling. Probably with a larger input or a different program we could see more interesting behavior.

```
CPU running our application:
procs -----------memory---------- ---swap-- -----io---- -system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so bi    bo   in   cs us sy id wa st
 0  0  0 23684480    212 5786964    0    0     0     6   24   20  3  0 97  0  0
top - 15:19:21 up 24 days, 23:48,  0 users,  load average: 0.16, 2.00, 8.07
Tasks: 552 total,   1 running, 551 sleeping,   0 stopped,   0 zombie
Cpu(s): 69.9%us,  0.7%sy,  0.0%ni, 29.4%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   32212M total,  2522M used,    29690M free,          0M buffers
Swap:  11442M total,  2237M used,    9205M free,    213M cached

  PID USER     PR  NI  VIRT  RES  SHR S   %CPU %MEM TIME+  COMMAND
 8905 sam14021  20   0 12768  820  484 S    0  0.0   0:00.02 1445606222.1306
 9285 sam14021  20   0 11212 1532  848 R    0  0.0   0:00.00 top
```

CPU not running our application (main script is doing a sleep):

```
procs -----------memory---------- ---swap-- -----io---- -system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so bi    bo   in   cs us sy id wa st
 0  0 2290712 30403244       0 218904       0      0      0      1      0      1 70  1 29
0  0
top - 15:19:21 up 1 day,  6:43, 36 users,  load average: 0.71, 1.53, 1.61
Tasks: 743 total,   1 running, 742 sleeping,   0 stopped,   0 zombie
Cpu(s):  2.8%us,  0.4%sy,  0.0%ni, 96.7%id,  0.0%wa,  0.0%hi,  0.1%si,  0.0%st
Mem:   32212M total,  9082M used,    23129M free,          0M buffers
Swap:  11442M total,         0M used,       11442M free,   5651M cached

  PID USER     PR  NI  VIRT  RES  SHR S   %CPU %MEM TIME+  COMMAND
26990 sam14021  20   0 11344 1680  848 R    2  0.0   0:00.02 top
24845 sam14021  20   0 12768  808  484 S    0  0.0   0:00.09 exercise_9.sh
32275 sam14021  20   0 81536 2148 1288 S    0  0.0   0:00.18 sshd
32276 sam14021  20   0 16440 3716 1784 S    0  0.0   0:00.27 bash
```

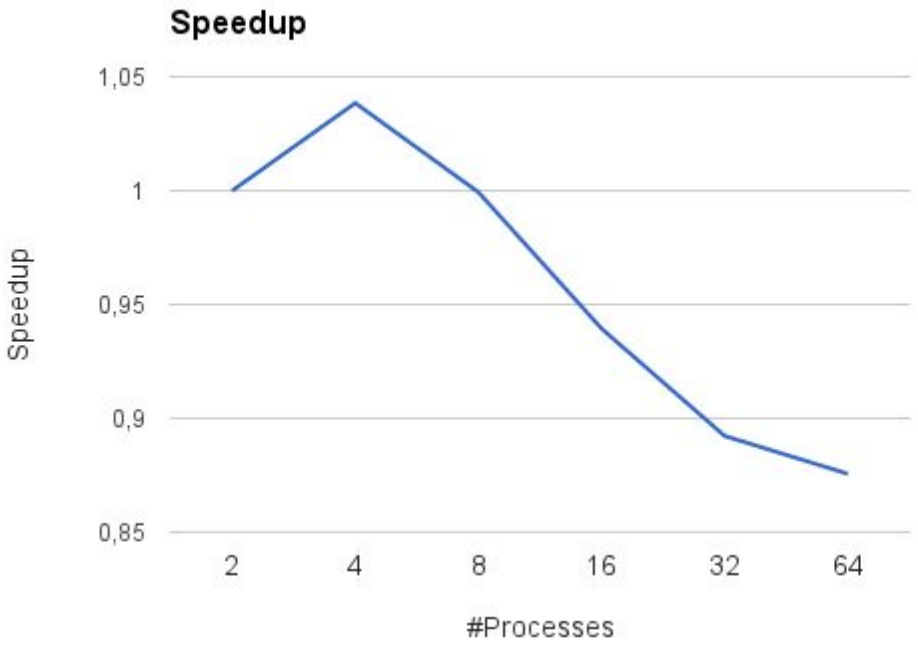
Exercise 10 (OPTIONAL): Include in your log file information extracted from vmstat and perf. Include in the answer your new LSF file.

```
$ cat exercise_9.sh
#!/bin/bash
#BSUB -J "SA-MIRI-log.file.generator"
#BSUB -n 16
#BSUB -o %J.out
#BSUB -e %J.err
#BSUB -W 00:15

LOG=profile.log
cont=0

# Launch vmstat for infinity into logfile
( while [ $cont -eq 0 ]; do
     top -n 1 -b -u $USER -H >> $LOG;
     vmstat >>$LOG ;
done ) &

sleep 1

echo "########################################" >> $LOG
echo "##### Starting execution ####" >> $LOG
echo "########################################" >> $LOG

perf stat mpirun -np 16 ./trapezoidal_mpi < input.txt >
normal_output.txt

echo "########################################" >> $LOG
echo "#### Stopping execution #####" >> $LOG
echo "########################################" >> $LOG

cont=1

sleep 1
```

Exercise 11: Now timing again the parallel MPI matrix-vector multiplication from exercise 10 in Hands-on 3 for 2, 4, 8, 16, 32 and 64. Divide the previous time obtained for each process in two: the serial part executed by process 0 and the parallel part. Compute the percentage.

	2 processess	4 processes	8 processes	16 processes	32 processes	64 processes
Serial	28,758	27,865	29,053	30,932	32,62	33,262
Parallel	0,392	0,205	0,117	0,098	0,05	0,028
% Serial	98,65523156	99,26968294	99,59890298	99,6841766	99,84695439	99,91589066
% Parallel	1,344768439	0,730317064 5	0,401097017 5	0,315823396 7	0,153045607 6	0,084109342 14
Total	29,15	28,07	29,17	31,03	32,67	33,29
% Total	100	100	100	100	100	100

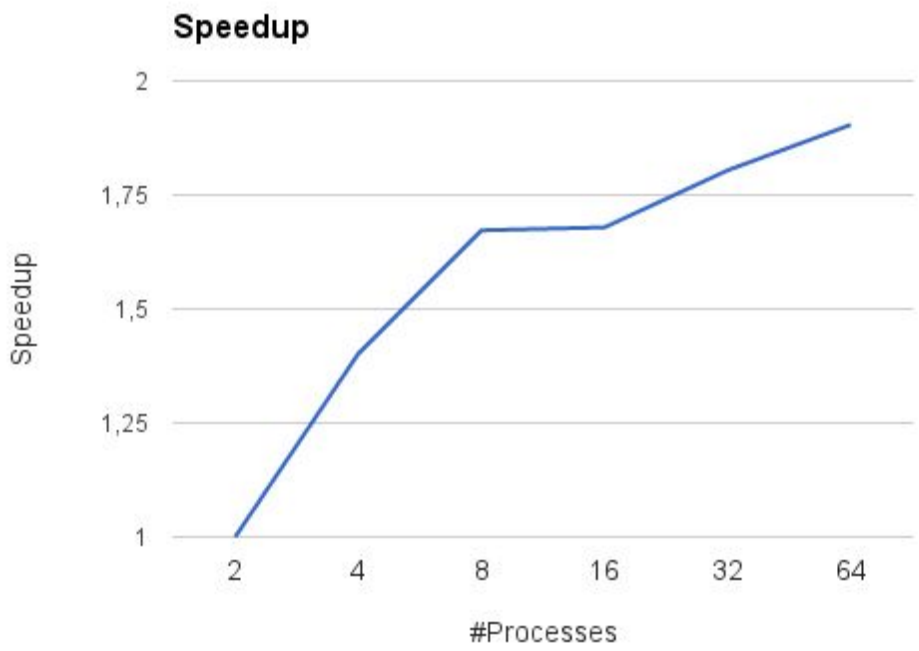
Exercise 12: Plot the results of the previous exercise in the form speedup vs number of processors. What are your conclusions?



In the plot we see that we are actually losing performance as we increase the number of processes working on the data set, and then if we look the table from exercise 11 we see that the parallel part of the program is very tiny compared to the serial part of the program, so actually we spend more time communicating (the serial part is increasing as the parallel part is decreasing)

Exercise 13: Compute again the following table with the new code and plot the results. Compare the results with the previous ones. What are your conclusions?

	2 processes s	4 processes	8 processes	16 processes	32 processes	64 processes
Serial	29,36	28,14	29,06	30,81	32,87	33,3
Parallel	38,69	20,4	11,63	9,72	4,85	2,44
% Serial	43,14474651	57,97280593	71,41803883	76,01776462	87,14209968	93,1729155
% Parallel	56,85525349	42,02719407	28,58196117	23,98223538	12,85790032	6,827084499
Total	68,05	48,54	40,69	40,53	37,72	35,74
% Total	100	100	100	100	100	100



Exercise 14 (OPTIONAL): Do the same as previous exercise with a “more more” parallel code with “x1000” in the matrix-vector multiplication. Plot and compare the results with the previous ones.

	2 processess	4 processes	8 processes	16 processes	32 processes	64 processes
Serial	31,06	28,5	29,44	31,08	32,93	32,79
Parallel	386,31	205,24	116,1	98,17	49,11	24,47
% Serial	7,441838177	12,193035	20,22811598	24,04642166	40,13895661	57,26510653
% Parallel	92,55816182	87,806965	79,77188402	75,95357834	59,86104339	42,73489347
Total	417,37	233,74	145,54	129,25	82,04	57,26
% Total	100	100	100	100	100	100