Assignment 1: Single Node Performance

Mihai-Gabriel Robescu (03709201) : mihai.robescu@tum.de

Hsieh Yi-Han (03708688) : kimihsieh.tw@tum.de

Yi JU (03691953) : yi.ju@tum.de

Koushik Karmakar (03704659): koushik.karmakar@tum.de

4. Performance Baseline

4.1 GNU Profiler

4.1.1 Required Submission Files

Ans. Please check the included file gprof.out in the attached solutions.zip.

4.1.2 Questions

1. Which routines took 80% or more of the execution time of the benchmark? (1 point)

Ans. CalcHourglassContralForElems(Domain&,double*,double) main

IntegratestressForElems

2. Is the measured execution time of the application affected by gprof? Hint: use the time command to determine this.(1 point)

Ans. 2 more second almost the same time

3. Can gprof analyze the loops (for, while, do-while, etc.) of the application?(1 point)

Ans. No *Gprof* count the time for self-defined functions

4. Is gprof adequate for the analysis of long running programs? Explain.(2 points)

Ans. Yes. But the *Gprof* just count the time of CPU. For example, when the function *time* is operated *Gprof* could not count its time.

5. Is gprof capable of analyzing parallel applications?(1 point)

Ans. Yes

6. What is necessary to analyze parallel applications?(2 points)

Ans. For Multiprocessing, we need to modify the name of *gmont.out ;* for multithreading, we need to call *setitimer* in each function spawned by a thread. We also can implement a wrapper for pthread-creat.

7. Were there any performance differences between the Intel compiler and the GNU compiler? (1 point)

Ans. For the GNU time = 51.570s
For Intel compiler time = 44.618s

4.2 Compiler Flags

4.2.1 Required Submission Files

1. *The script that automates the evaluation.(5 points)*

Ans. Please check the attached solutions.zip file for *automat_GNU, automat_intel files and batch_serial.sh* files.

2. *Separate Makefiles that contain the best combination of parameters for the Intel and GNU compilers.(1 point)*

Ans. Please check the attached solutions.zip for *automat_GNU and automat_Intel* makefiles.

4.2.2 Questions

1. Look at the compilers' help (by issuing icc -help and gcc -help). How many opti- mization flags are available for each compiler (approximately)?(1 point)

Ans. About 150 flags for GCC About 550 flags for icc-

2. Given how much time it takes to evaluate a combination of compiler flags, is it realistic to test all possible combinations of available compiler flags? What could be a possible solution? (2 points)

Ans. Each run takes about 1 or 2 min. For the flags we given, it is possible. Because the number of rounds that the evaluation of GNU takes is:

$$n = \sum_{i=1}^{7} {}^{7}C_i = 127$$

The number of rounds that the evaluation of Intel compiler takes is:

$$n = \sum_{i=1}^{4} {}^{4}C_i = 15$$

So the needed time is under control. But it is almost impossible to do evaluation of all combinations of compiler flags. The number of rounds that the evaluation of k flags is:

$$n = \sum_{i=1}^{k} {}^{k}C_i = 2^k - 1$$

The number increases exponentially.
We could separate the task and use several computers to do the test. We also could use the information provided by the developers and delete the combination of the flags for totally opposite aims.

3. Which compiler and optimization flags combination produced the fastest binary?(1 point)

Ans.The best combination of optimisation flags for intel compiler is:
-march=native -xHost -unroll.
The run needs 50.77s.
The best combination of optimisation flags for GNU compiler is :
-march=native -fomit-frame-pointer -floop-block -floop-interchange -floop-strip-mine

-funroll-loops.
The run needs 52.27s.
The intel compiler with -march=native -xHost -unroll produced fastest binary.

4.3. Optimization Pragmas

4.3.1. Submit the modified source file with the added #pragma annotation. (2 points)

Ans. Check the source file lulesh_4_3_pragma.cc included in the solutions.zip.

4.3.2 Questions

1. What is the difference between Intel's simd, vector and ivdep #pragma directives? (2 points)

Ans. These three #pragma allow compiler to vectorize FOR LOOP in different way. When compiler try to vectorize the vectors, compiler see assumed dependence as proven dependence, which prevents vectorization. *ivdep* makes compiler ignore assumed dependence, but not proven dependence. However *simd* might ignore proven dependence. Then force compiler to vectorize the loop, because the proven dependence is ignored. vector tries vectorize the loop even if losing performance snd vector holds assertion of independency based on different keywords.

2. Why did you choose to apply the selected #pragma in the particular location?(2 points)

Ans. I add #pragma *simd* (intel compiler) before CalcHourglassControlForElems. The functions I add #pragma have majority of execute time based on 4.1 and have FOR LOOP. For accelerate computing, we try to vectorize FOR LOOP to decrease the number of instruction. That's is the reason.

4.4 Inline Assembler

4.4.1 Required Submission Files

Q. Submit the pair of matching code snippets with Intel and AT&T styles.(2 points)

Ans. Intel Assembly language vs. AT&T Syntax [1]

Intel Syntax	AT&T Syntax
`instr foo,segreg:[base+index*scale+disp]`	`instr %segreg:disp(base,index,scale),foo`
`mov eax,[ebx+20h]`	`movl 0x20(%ebx),%eax`
`add eax,[ebx+ecx*2h`	`addl (%ebx,%ecx,0x2),%eax`
`lea eax,[ebx+ecx]`	`leal (%ebx,%ecx),%eax`
`sub eax,[ebx+ecx*4h-20h]`	`subl -0x20(%ebx,%ecx,0x4),%eax`

4.4.2 Questions

1. Is the inline assembler necessarily faster than compiler generated code?(2 points)

Ans. No, high level language like C++ can generated faster codes than a low-level language like assembly. It can also depend on the experience of the assembler programmer, and on the fact that some compilers don't know to take full advantage of the resources like CPU and in those cases, the code generated by the programmer is faster.

2. On the release of a CPU with new instructions, can you use an inline assembler to take advantage of these instructions if the compiler does not support them yet?(1 point)

Ans. Yes, with the assembler you program the hardware directly and so you can take advantage of the instructions.

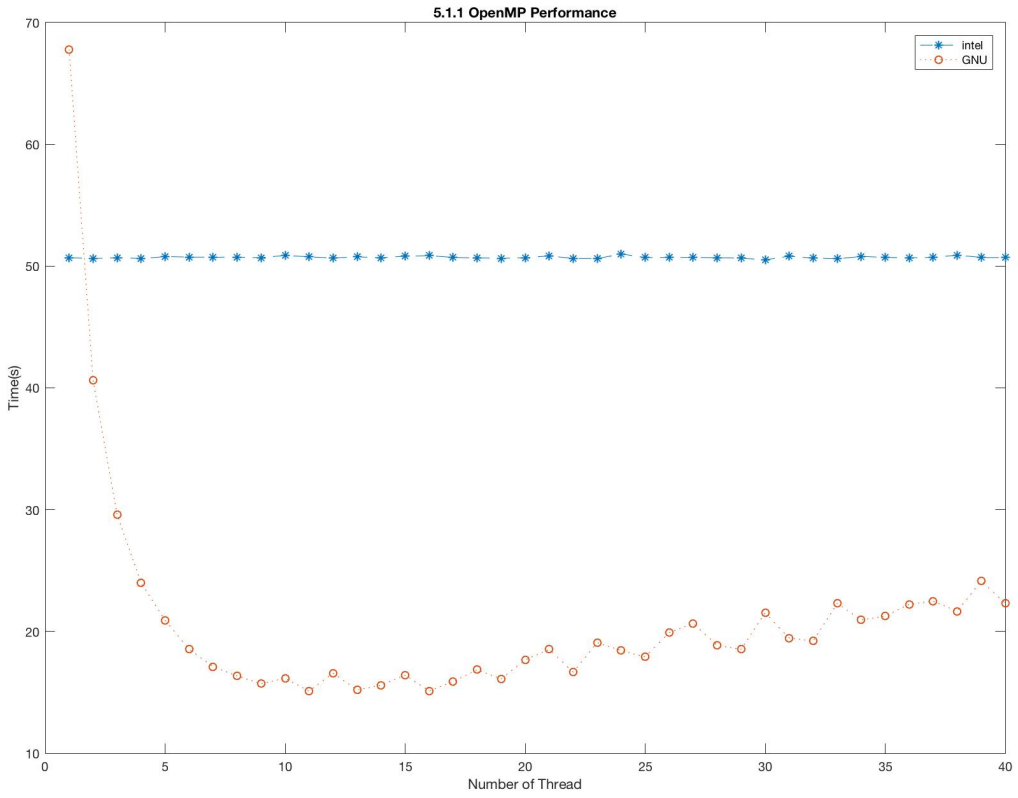
3. What is AVX-512? Which CPUs support it? Is there any compiler or language support for these instructions at this moment?(1 point)

Ans. It is an Advanced Vector Extension which allows the extension of the register size to 512 bits. It does support CPUs like Coffee Lake processor Q4 20, Skylake-X processor Q2 2017, Broadwell E processor Q2 2016.Yes, there compilers like GCC, Clang or Java 9 that support it. [2]

5. Performance Scaling

5.1 OpenMP

5.1.1 Include the plots for both the Intel and the GNU compilers in the report.



Q. Copy the following information from your job script for each test:

Test 1:

node =
total_tasks =
OMP_Threads

Test 1: time = 50.69 s node = 1 OMP threads= 1	Test 2: time = 50.62 s node = 1 OMP threads= 2	Test 3: time = 50.69 s node = 1 OMP threads= 3
Test 4: time = 50.62 s node = 1 OMP threads= 4	Test 5: time = 50.78 s node = 1 OMP threads= 5	Test 6: time = 50.74 s node = 1 OMP threads= 6
Test 7: time = 50.73s node = 1 OMP threads= 7	Test 8: time = 50.74 s node = 1 OMP threads= 8	Test 9: time = 50.69 s node = 1 OMP threads= 9
Test 10: time = 50.88 s node = 1 OMP threads= 10	Test 11: time = 50.76 s node = 1 OMP threads= 11	Test 12: time = 50.67 s node = 1 OMP threads= 12
Test 13: time = 50.76 s node = 1 OMP threads= 13	Test 14: time = 50.69 s node = 1 OMP threads= 14	Test 15: time = 50.81 s node = 1 OMP threads=15
Test 16: time = 50.86 s node = 1 OMP threads= 16	Test 17: time = 50.71 s node = 1 OMP threads= 17	Test 18: time = 50.67s node = 1 OMP threads= 18
Test 19: time = 50.64 s node = 1 OMP threads= 19	Test 20: time = 50.84 s node = 1 OMP threads= 20	Test 21: time = 50.84 s node = 1 OMP threads= 21
Test 22: time = 50.63 s node = 1 OMP threads= 22	Test 23: time = 50.62 s node = 1 OMP threads= 23	Test 24: time = 50.99 s node = 1 OMP threads= 24

Test : 25: time = 50.71 s node = 1 OMP threads= 25	Test 26: time = 50.71 s node = 1 OMP threads= 26	Test 27: time = 50.72 s node = 1 OMP threads= 27
Test 28: time = 50.69 s node = 1 OMP threads= 28	Test 29: time = 50.65s node = 1 OMP threads= 29	Test 30: time = 50.52s node = 1 OMP threads= 30
Test 31: time = 50.81 s node = 1 OMP threads= 31	Test 32: time = 50.81 s node = 1 OMP threads= 32	Test 33: time = 50.81 s node = 1 OMP threads= 33
Test 34: time = 50.79 s node = 1 OMP threads= 34	Test 35: time = 50.72 s node = 1 OMP threads= 35	Test 36: time = 50.68 s node = 1 OMP threads= 36
Test 37: time = 50.72 s node = 1 OMP threads= 37	Test 38: time = 50.90 s node = 1 OMP threads= 38	Test 39: time = 50.71 s node = 1 OMP threads= 39
Test 40: time = 50.70 s node = 1 OMP threads= 40		

Provide a description of the type of scaling (weak/strong) achieved for one of the compilers.

Ans. Both scaling are weak achieved. After adding more threads, the performance are not scaling in intel compiler. For GNU compiler, it seems scaling are strong achieved when the numbers of threads less than 11 but after 11, the performance even decay slightly.

5.1.2 Question

1. *Was linear scalability achieved?*

Ans. No. Based on the plot, linear scalability are not achieved in both case.

2. *On which thread-count was the maximum performance achieved? Was it the same for both the Intel and the GNU compilers?(3 points)*

Ans. For GNU compiler, the maximum performance is achieved when numbers of thread equal are 11.

Ans. No, the performance in intel compiler is almost the same. There is no obvious local minimum point in the plot.

5.2.2

4. How does the performance compare to the results achieved with OpenMP in Section 5.1?(2 points)

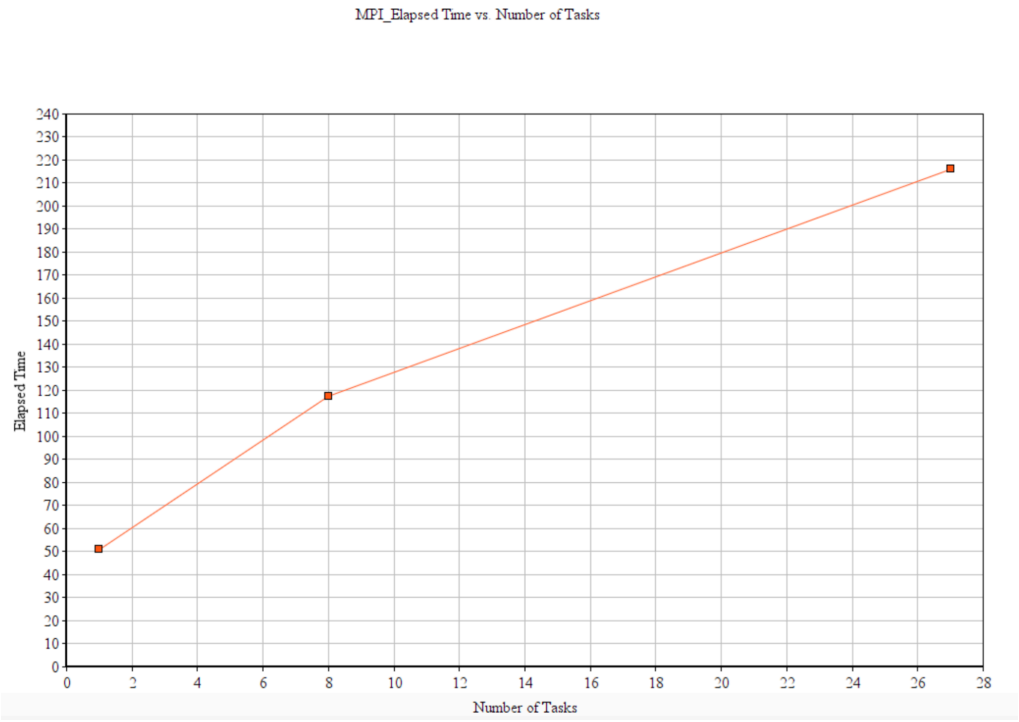
Ans. The performance in Section 5.1 with OpenMP are both better a lot than Section 5.2 with MPI.

5.2 MPI

5.2.1. Questions

1. Include the plots for the Intel compiler in the report.

Ans.



2. Copy the following information from your job script for each test:

Test 1:
node =
total_tasks =

Ans.

Test 1: time = 50.91 s node = 1 total_tasks= 40	Test 4: time = 111.23 s node = 4 total_tasks= 8	Test 7: time = 191.66 s node = 4 total_tasks= 27
Test 2: time = 117.49 s node = 1 total_tasks= 8	Test 5: time = 215.97 s node = 1 total_tasks= 27	
Test 3: time = 109.83 s node = 2 total_tasks= 8	Test 6: time = 195.99 s node = 2 total_tasks= 27	

3. Provide a description of the type of scaling (weak/strong) achieved for the Intel compiler.

Ans. The scaling achieved with the intel compiler is weak scaling since the solution time varies with the number of processors for a fixed problem size per processor. If we keep increasing the no. of nodes, the elapsed time doesn't change after the number of tasks exceed a certain threshold. For the above data provided, the scaling isn't really consistent.

5.2.2. Questions

1. What are the valid combinations of processes allowed?(2 points)

Ans. Valid combinations of processes allowed are cubes of a number and since the maximum no. of cores is 40, it also has to be less than 40. So, the possible options are - 1, 8 and 27.

2. Was linear scalability achieved?(2 points)

Ans. Linear Scalability was achieved for most tests since increasing the no. of nodes doest result in the execution time being reduced for most of the tests.

3. On which process-count was the maximum performance achieved? (3 points)

Ans. For number of processes=1, the maximum performance was achieved i.e. 50.91 (s).

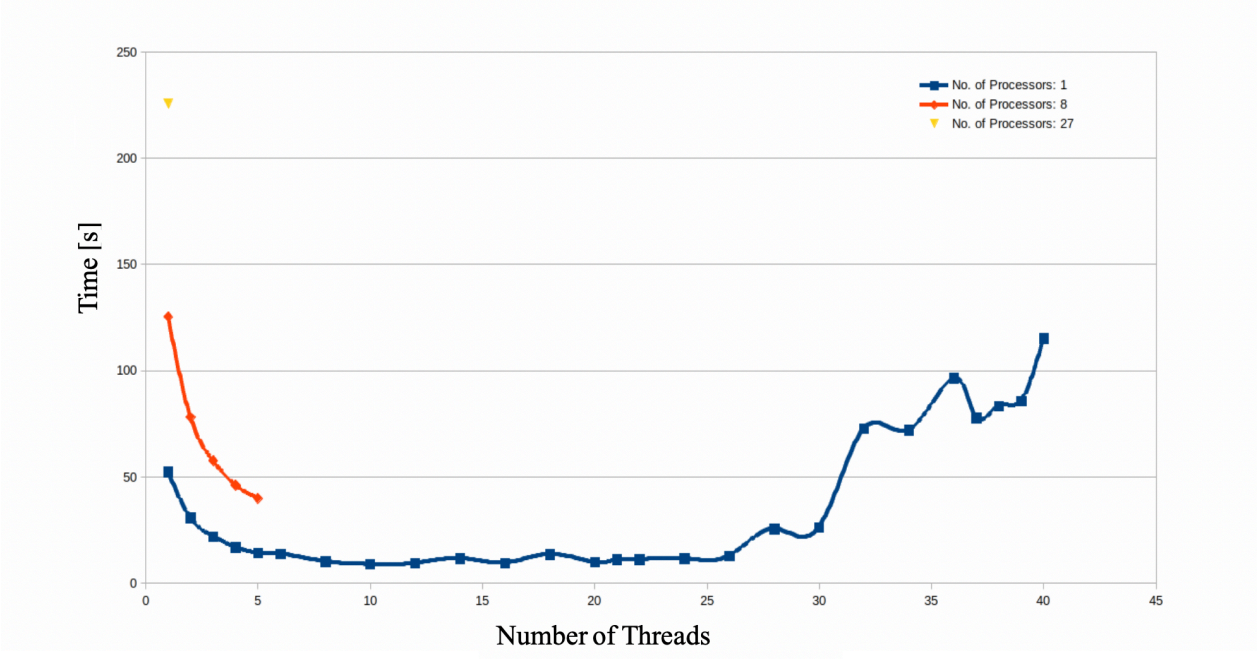
4. How does the performance compare to the results achieved with OpenMP in Section 5.1?(2 points)

Ans. The performance results achieved with MPI is worse when compared to OpenMP for equivalent no. of tasks and nodes.

5.3 MPI + OpenMP

5.3.1 Required Submission Files (13 points)

1. Include the plots for the Intel compiler in the report.



2. Copy the following information from your job script for each test:

Test 1: time = 39.98 s node = 1 total_tasks= 40 OMP threads= 5	Test 9: time = 83.45 s node = 1 total_tasks= 38 OMP threads= 38	Test 17: time = 11.69 s node = 1 total_tasks= 24 OMP threads= 24
Test 2: time = 46.12 s node = 1 total_tasks= 32 OMP threads= 4	Test 10: time = 77.83 s node = 1 total_tasks= 37 OMP threads= 37	Test 18: time = 11.22 s node = 1 total_tasks= 22 OMP threads= 22

Test 3: time = 57.75 s node = 1 total_tasks= 24 OMP threads= 3	Test 11: time = 96.52 s node = 1 total_tasks= 36 OMP threads= 36	Test 19: time = 11.10 s node = 1 total_tasks= 21 OMP threads= 21
Test 4: time = 78.10 s node = 1 total_tasks= 16 OMP threads= 2	Test 12: time = 72.16 s node = 1 total_tasks= 34 OMP threads= 34	Test 20: time = 9.93 s node = 1 total_tasks= 20 OMP threads= 20
Test 5: time = 125.47 s node = 1 total_tasks= 8 OMP threads= 1	Test 13: time = 72.72 s node = 1 total_tasks= 32 OMP threads= 32	Test 21: time = 13.61 s node = 1 total_tasks= 18 OMP threads= 18
Test 6: time = 225.74 s node = 1 total_tasks= 27 OMP threads= 1	Test 14: time = 26.17 s node = 1 total_tasks= 30 OMP threads= 30	Test 22: time = 9.58 s node = 1 total_tasks= 16 OMP threads= 16
Test 7: time = 115.16 s node = 1 total_tasks= 40 OMP threads= 40	Test 15: time = 25.57 s node = 1 total_tasks= 28 OMP threads= 28	Test 23: time = 11.71 s node = 1 total_tasks= 14 OMP threads= 14
Test 8: time = 85.72 s node = 1 total_tasks= 39 OMP threads= 39	Test 16: time = 12.82 s node = 1 total_tasks= 26 OMP threads= 26	Test 24: time = 9.43 s node = 1 total_tasks= 12 OMP threads= 12
Test 25: time = 8.92 s node = 1 total_tasks= 10 OMP threads= 10	Test 28: time = 14.32 s node = 1 total_tasks= 5 OMP threads= 5	Test 31: time = 30.77 s node = 1 total_tasks= 2 OMP threads= 2

Test 26: time = 10.26 s node = 1 total_tasks= 8 OMP threads= 8	Test 29: time = 16.97 s node = 1 total_tasks= 4 OMP threads= 4	Test 32: time = 52.46 s node = 1 total_tasks= 1 OMP threads= 1
Test 27: time = 13.89 s node = 1 total_tasks= 6 OMP threads= 6	Test 30: time = 22.00 s node = 1 total_tasks= 3 OMP threads= 3	

5.3.2 Questions

1. What are the valid combinations of processes and threads?(3 points)

Ans. The Number of processes has to be a cubic number like 1, 8 or 27, which means that the number of threads is the integer part of 40/(number of processes), as 40 is the total number of cores in the node. That means for 27 processes you can have only one thread, for 8 processes one can have from 1 to 5 threads and for just 1 process one can have from 1 to 40 threads.

2.Was linear scalability achieved?(2 points)

Ans. No, linear scalability was not achieved as after a number of threads the execution time increased.

3. On which combination of processes and threads was the maximum performance achieved? (3 points)

Ans. The best combinations between processes is of one processes with a number of threads from 5 to 16.

4. How does the performance compare to the results achieved with OpenMP in Section 5.1 and with MPI in Section 5.2?(4 points)

Ans. OpenMP + MPI is in between of both. If it has more execution threads then it might be faster then OpenMP, but OpenMP with more threads is the fastest.

5. Which solution is overall the fastest?(5 points)

Ans. OpenMP

6.Would you have guessed this best combination before performing the experiments in Sections 5.1, 5.2 and 5.3?(2 points)

Ans. No.

6. Work Breakdown Structure

P = **P**articipate **R** = be **R**esponsible to **A** = **A**bsent

Task Name	4.1	4.2	4.3	4.4	5.1	5.2	5.3	Report
Mihai-Gabriel Robescu	P	P	P	R	P	R	R	P
Hsieh Yi-Han	P	P	R	P	R	P	R	P
Yi JU	P	R	P	P	R	P	P	R
Koushik Karmakar	A	A	P	P	P	R	P	R