Programming of Supercomputers
Worksheet 1

Oleksandr Voloshyn <o.voloshyn@tum.de>
Qunsheng Huang <keefe.huang@tum.de>
Tommaso Bianucci <bianucci@in.tum.de>

November 11, 2018

Group members's contributions

Oleksandr Voloshyn: worked on 4.1, 4.2, 4.3, 5.1, 5.3
Tommaso Bianucci: worked on 4.1, 4.2, 4.3, 5.2, 5.3
Qunsheng Huang: worked on 4.1, 4.3, 4.4, 5.1

1 Performance baseline

1.1 GNU Profiler

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

   (a) Serial

   - `EvalEOSForElems()` — 28.18%
   - `CalcHourglassControlForElems()` — 17.10%
   - `CalcFBHourglassForceForElems()` — 15.76%
   - `CalcKinematicsForElems()` — 11.84%
   - `IntegrateStressForElems()` — 10.98%

   These functions account for 83.86% of total execution time.

   (b) OpenMP

   - `CalcHourglassControlForElems()` — 29.90%
   - `ApplyMaterialPropertiesForElems()` — 22.58%
   - `CalcFBHourglassForceForElems()` — 15.84%
   - `IntegrateStressForElems()` — 14.28%

   These functions account for 82.60% of total execution time.

   (c) MPI

   - `EvalEOSForElems()` — 24.44%
   - `LagrangeNodal()` — 21.86%

  - `CalcFBHourglassForceForElems()` — 16.67%
  - `CalcKinematicsForElems()` — 11.12%
  - `IntegrateStressForElems()` — 10.15%

  These functions account for 84.24% of total execution time.

(d) Hybrid

  - `CalcFBHourglassForceForElems()` — 23.05%
  - `EvalEOSForElems()` — 21.22%
  - `LagrangeNodal()` — 18.95%
  - `IntegrateStressForElems()` — 13.02%
  - `CalcKinematicsForElems()` — 10.29%

  These functions account for 86.53% of total execution time.

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

   The measured execution time does not differ with use of gprof. This was tested with the four benchmarks with the following results:

	Time without Gprof (s)	Time with Gprof (s)
Serial	53.90	53.88
OpenMP	21.06	20.95
MPI	120.78	120.84
Hybrid	46.9	45.95

   Each benchmark was run 2-3 times with and without gprof. The times did vary slightly but indicated that the usage of gprof did not change th execution time. The results in the above table can be seen in the attached gprof files.

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

   No, gprof cannot be used to analyse loops.

4. Is gprof capable of analyzing parallel applications?

   Yes, gprof can be used to analyse some parallel applications. However, additional steps must be taken for proper analysis. Otherwise, some information will be obscured:

   - In the case of MPI implementations, only the performance of the main thread is recorded.
   - In the case of OpenMP processes, much of the profiling information is obscured—as seen in the attached gprof, the code spends more than 50% of the time in "frame dummy", a function that does not exist in source code.

5. What is necessary to analyze parallel applications?

   Gprof typically only produces one `gmon.out` output file for the main process. However, in a parallel process, multiple output files are required. When using MPI processes, it is possible to inform gprof, via environmental variables, of the number of threads produced and run. One `gmon.out` is then produced per thread. These files can then be aggregated to determine the overall behavior of all threads together or analysed separately. For OpenMP applications, the tag `-openmp-stubs` must be used, which will force a sequential execution of the program for profiling purposes.

6. Where there performance differences between the GNU and the Intel compiler?

   There were differences between the two compilers. In general, the code compiled by the Intel compiler was slightly faster. The difference between the code compiled by the two compilers is approximately 2s and this varied slightly each job. Four jobs were run using the automated script in Section 4.2 of the assignment—overall, the Intel compiled code ran approximately 2s faster than the GNU, both with the optimal flags.

1.2 Compiler flags

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

   `gcc --help=optimizer` shows approximately 100 compiler flags, `icpc --help <category>` show approximately 18 optimization flags (`opt`), 8 interprocedural optimization flags (`ipo`), 60 advanced optimization options (`advanced`) and 20 profile guided optimizations (`pgo`), for a total of 106 flags.

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?

   Each test run takes approximately 40-50 seconds to complete, excluding the time required for compilation of the code. Given that there are more than 100 optimization flags for each compiler, we can conservatively estimate that the total time to completion would exceed $6 \times 10^{32} s$, which is completely unfeasible. Instead, it would make sense for us to examine the code to determine what possible bottlenecks exist and apply flags that could improve the situation.

3. Which compiler and optimization flags combination produced the fastest binary?

   For the GNU compiler, we determined that the effect of the flags was somewhat inconsistent. We performed a simple statistical analysis of the effects of the flags over 4 runs. Overall, we observed that the flags

"-march=native" and "-funroll-loops" were constantly associated with binaries with the best performance.

For the Intel compiler, the fastest binary only used "-unroll" flag with a time of 50.43s to completion with a speedup of 1.00258. The second fastest binary was without flags with a time of 50.56.

The fastest binary from both compilers was compiled using the Intel compiler with the "-unroll" flag.

1.3 Optimization pragmas

1. What is the difference between Intel's *simd*, *vector* and *ivdep* *#pragma* directives?

   The three pragmas allow the compiler to ignore certain requirements when vectorizing. The simd pragma forces vectorization to occur, ignoring safety or cost. The vector pragma forces the compiler to ignore cost when deciding to vectorize (as a result the code may run slower after vectorization). The ivdep pragma instructs compiler to ignore assumed vector dependencies—i.e. the compiler may choose to treat assumed dependencies as proven dependencies. This pragma instructs compiler to ignore such assumptions and proceed with vectorisation.

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

   We chose to apply `#pragma loop_count avg(27000)` to the function `CalcHourglassControlForElems` in lulesh.cc. When running the task in serial, more than 70% of overall runtime time was spent in this function. This function contains a highly complicated loop that may require specific treatment for optimization.

1.4 Inline assembler

1. Is the inline assembler necessarily faster than compiler generated code?

   Not necessarily. The speed increase is dependent on the implementation on the code—especially when modern compilers already implement many forms of optimization. It is often likely that hand-written assembler code is often slower than compiler optimized assembler code. However, there do exist specific cases where an inline assembler may be faster. The most relevant cases to this module include:

   - Implement instructions which are not yet supported by the current compiler. For example, a full-multiplication operator (2N-bit result from N-bit inputs), which is not available in C . Most x86 CPUs can, however, perform this operation in a single instruction. In these rare cases, one can use inline assemblers to access optimal functions that belong to the specific CPU when available.

- Spot-optimizing specific lines of code. For example, compilers may not optimally vectorize complicated loops, since it has to deal with many general cases. If the programmer understands the problem well, it may be beneficial to use the inline assembler to perform the loop operations.

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?

   Yes. One benefit to inline assemblers is that it allows access to processor specific instructions which the compiler may not support. Examples may include FPU instructions that may be faster than compiler generated floating operations. However, the onus of correct usage of these commands then falls on the programmer.

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

   AVX stands for Advanced Vector Extensions SIMD instructions. AVX-512 are the 512-bit extensions to the preexisting AVX/AVX2 instructions. They were first supported by the Xeon-Phi x200 (Knights Landing) and Skylake-X CPUs. Note that AVX-512 is not the first 512-bit SIMD instructions released by Intel. These commands are supported in the intel C++ compiler under the namespace AVX. The `-xCOMMON-AVX512` flag can be used for auto-vectorization using AVX-512 functions.

2 Performance Scaling

2.1 OpenMP

1. Was linear scalability achieved?

   Linear scalability was not achieved. However, for the GNU compiler, we see that speedup increases between 1 and 5 threads. We see a similar pattern for the Intel compiler between 1 and 7 threads. These two trends can be seen in Fig.1. After the point of peak speedup, we then see that performance worsens and speedup values decrease as the costs of parallelization begin to dominate the gains from additional threads. This is especially the case for the Intel compiler—when using more than 26 processors, the speedup drops below 1.

   We also see that–when increasing the number of threads while keeping the domain constant–the performance of the program cannot be guaranteed. We observe sudden decreases in speedup with both compiler above 8 threads (seen as spiky segments along the speedup graph)—this is may be due to issues with non-uniform memory access.

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

   For the Intel compiler, the maximum speedup was 4.64 (11.32s) with 16 threads. For the GNU compiler, the maximum speedup was 3.43 (19.57s) with 15 threads. The minimum point can be seen in Fig.1.b. While the Intel compiler experienced the best speedup (and fastest time), the speedup also rapidly decayed after a certain point (at 40 processors, the program took almost 100s to complete). The program compiled by the GNU compiler did also see a reduction in speedup but the overall time remained below 30s at 40 processors.

2.2 MPI

1. What are the valid combinations of processes allowed?

   Only number of processes which are perfect cubes are allowed. So on a single node, which has 40 cores, we can use 1, 8, 27.

2. Was linear scalability achieved?

   Weak Ideal linear weak scaling occurs when the runtime does not vary with the number of processes. We could instead see (Fig.2.a) a marked increase in runtime in this case and the *weak parallel efficiency*[1] decreasing drastically.

   Strong Ideal linear strong scaling is achieved if the speedup equals the number of processes, i.e. if $S(N) = N$.

   To test strong scaling we adjusted the problem size per MPI process in order to keep the volume of the entire domain constant. This means that in our test runs with initial size 24, we used $s = 24$, $s = 12$, $s = 8$ for 1, 8 and 27 MPI processes respectively, while for our test runs with initial size 30, we used $s = 30$, $s = 15$, $s = 10$ for 1, 8 and 27 MPI processes respectively.

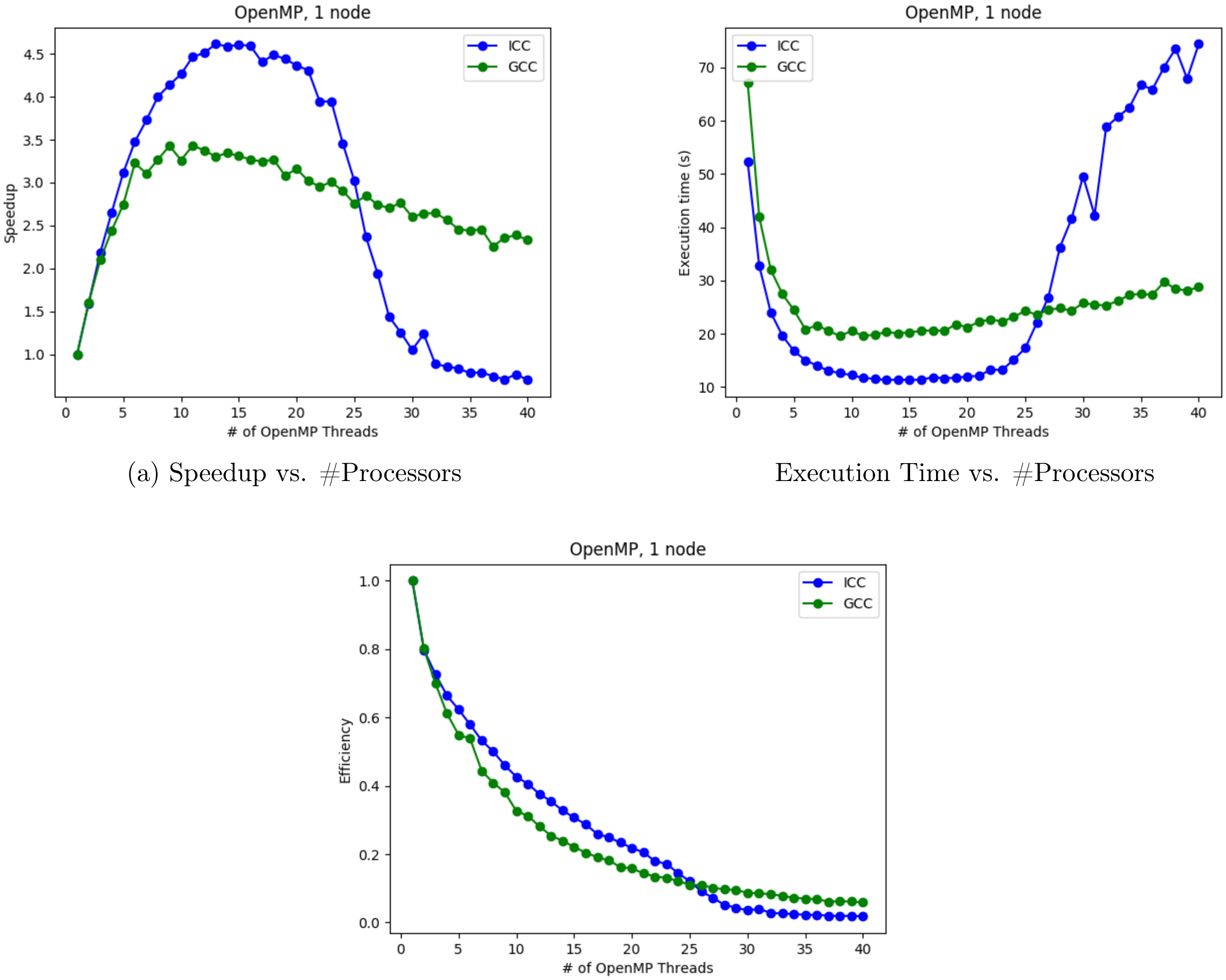
   We could observe an almost linear scaling behaviour, although not perfect. As shown in Fig.2.b, the actual speedup (solid blue line) quite follows the ideal linear scaling one (dashed red line). The *strong parallel efficiency*[2] drops to 80% with 27 MPI processes on the same node.

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

   From a strong scaling perspective, the best performance was achieved with 27 MPI processes.

[1] Defined as $speedup(N) = \frac{T(1)}{T(N)}$ where N is the number of processes.

[2] Defined as $speedup(N)/N = \frac{T(1)}{N \cdot T(N)}$.



(a) Speedup vs. #Processors

Execution Time vs. #Processors

(c) Efficiency vs. #Processors.

Figure 1: Performance statistics for OpenMP.

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

   The results clearly show that using MPI allows for a better strong scaling behaviour compared to OpenMP. With OpenMP we achieve a maximum speedup around 4.5, both with gcc and intel compilers, while with MPI we reach a speedup of 21 with 27 processes.

2.3 OpenMP + MPI

1. What are the valid combinations of processes and threads?

   Understanding that the multiple of MPI processes and OpenMP threads must be less than 40 and that the number of processes must increase in a cubic fashion, we have the following possible combinations for a single node:

   - MPI Processes: 8
     OpenMP Threads: 1-5
   - MPI Processes: 27
     OpenMP Threads: 1

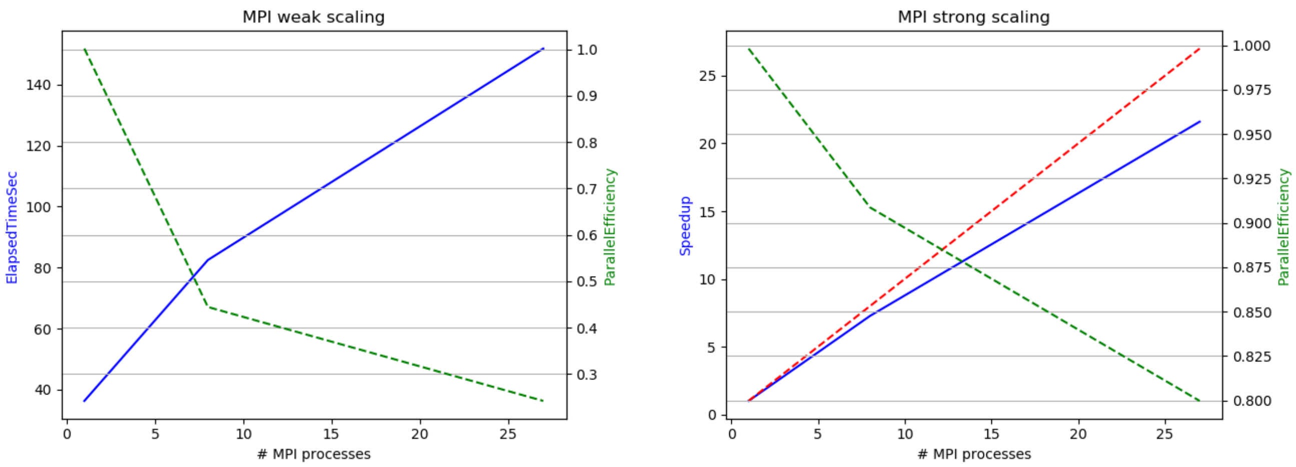
   We omit the study of when MPI processes = 1 as the scaling should be identical to the strong scaling in the OpenMP study.

2. Was linear scalability achieved?

   Weak scaling is achieved when we increase the domain size proportional to the number of processes launched. Currently, if we limit our experiments to a single node, we cannot examine any interesting cases for weak scaling. We would need to keep the number of OpenMP threads per MPI process constant while increasing the domain size to scale with the number of MPI processes. The only possible case would be to examine 1, 8 , and 27 MPI processes with 1 OpenMP thread—the performance is identical to the weak scaling analysed in question 5.2.

   Strong scaling is achieved by keeping the problem size constant and increasing the number of threads.This is seen when we run with 8 MPI processes and 1-5 OpenMP threads.

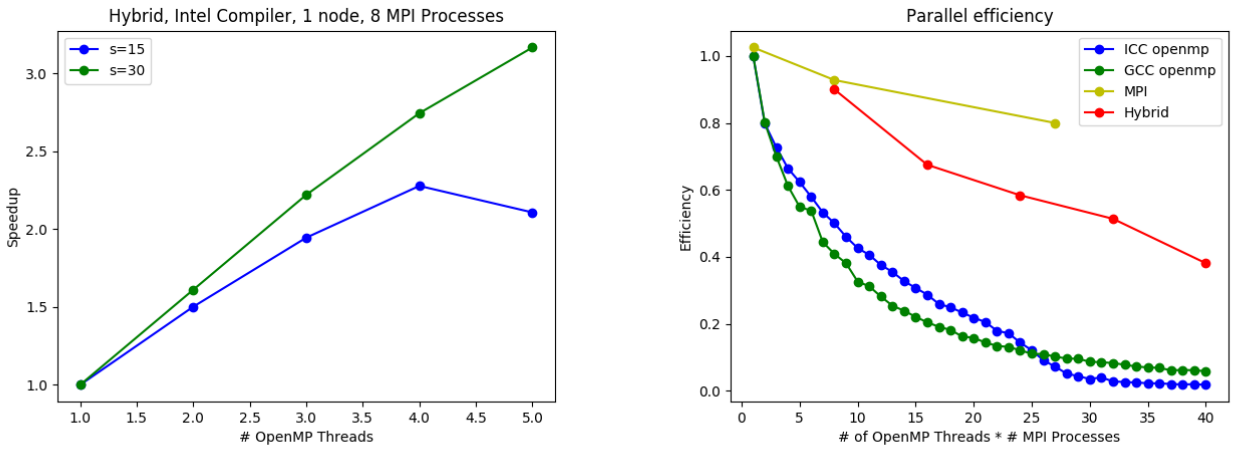
   Strong We ran 2 experiments, one with -s 15 and one with -s 30. We do not achieve linear scaling for either case. However, for the smaller domain size, we see in Fig.2.a that the speedup decreases when the number of OpenMP threads is increased to 5. This is expected because, per Amdahl's law, the maximum speedup for strong scaling is limited by problem size. At 5 OpenMP threads, the cost of parallelisation outweighs its benefits and we see a decrease in speedup. This can be remedied by increasing the overall domain size. We run the same form of strong scaling with $s = 30$ in and we see that speedup continues does not decrease with 5 OpenMP threads.



(a) Weak scaling

(b) Strong scaling

Figure 2: MPI scaling results. Results are averaged between runs with $s = 24$ and $s = 30$ as base sizes.



(a) Hybrid strong scaling, $s = 15$ and $s = 30$.

(b) Parallel efficiency comparison, $s = 30$.

Figure 3: Hybrid strong scaling results: comparison of different domain sizes and against other methods.

3. How does the performance compare to the results achieved with OpenMP in Section 5.1 and with MPI in Section 5.2?

   We see that the solutions for the hybrid (where number of OpenMP threads and MPI processes are greater than 1) are slower than a pure MPI implementation with 27 processes but faster than a pure OpenMP implementation with 15 threads. The highest speedup value was approximately 16.7 times or an run time of 3.75s.

4. Which solution is overall the fastest?

   Overall, the fastest solution is with a pure MPI implementation with a speedup of 21 and a runtime of 2.43 seconds. We can see that pure MPI approach scales much better with total number of processes (# MPI processes * # OpenMP threads) in Fig.2.b.

5. Would you have guessed this best combination before performing the experiments in Sections 5.1, 5.2 and 5.3?

   This is somewhat unexpected. We thought that the hybrid case should be able to outperform both MPI and OpenMP implementations. However, it appears that a pure MPI process is faster. We limited our experimentation to a single node due to the queue time of multi-node jobs. It is possible that at larger domain sizes, the hybrid approach will scale better.