Assignment 1:
Single Node Performance

Emily Bourne (emily.bourne@tum.de)
Abraham Duplaa (abraham.duplaa@tum.de)
Jiho Yang (jiho.yang@tum.de)

November 20, 2017

1 Performance Baseline

1.1 GNU Profiler

1.1.1 Questions

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

   The following functions calls took approximately 80% of the execution time: **main** (39.64%), **CalcHourglassControlForElems** (30.64%), and **IntegrateStressForElems** (10.26%) on *Phase 1* node, and **main** (35.41%), **CalcHourglassControlForElems** (34.49%) **CalcKinematicsForElems** (9.72%) on *Phase 2* node.

2. **Is the measured execution time of the application affected by gprof?**

   From our analysis shown in Table 1, gprof affected the execution time. However in this case, the execution time increased insignificantly. gprof increased Real time in both *Phase 1* and *Phase 2* by 0.1 seconds and 0.4 seconds, respectively. Execution time could be greater affected by gprof if more gprof options are implemented.

Node	With/without gprof	Real time	User time	System time
Phase 1	Without	40.304	36.294	3.964
Phase 1	With	40.411	36.662	3.692
Phase 2	Without	34.014	29.258	4.636
Phase 2	With	34.466	30.002	4.340

Table 1: Comparison of execution time with/without gprof. Execution times are obtained with "time" command

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

   Gprof does not provide an explicit method to analyze loops.

   However, there are a few flag options that can help in analyzing loop performance. When the '-A' flag is used, it produces a list of each function labelled with the number of times the function was called. Also, using 'gcc ... -g -pg -a' augments the program with basic-block counting code, enabling gprof to determine how many times each line of code was executed. Furthermore, gprof's '-l' option causes the program to perform line-by-line histogram samples.

   These types of flags could be used for a very rudimentary analysis of loops.

4. **Is gprof adequate for the analysis of long running programs? Explain.**

   Although a long running program would provide gprof with more sample counts and therefore more data points to analyze the code, gprof has several disadvantages that could make gprof unsuitable for long running and complex programs. Since gprof is not able to explicitly analyze loops, analyze cache misses, and some other important characteristics that should be analyzed, it may not provide enough information for a proper analysis.

5. **Is gprof capable of analyzing parallel applications?**

   Gprof is capable of analyzing parallel applications. Please see below for more information on how to analyze a parallael application.

6. **What is necessary to analyze parallel applications?**

   Only a few changes are needed to properly analyze a parallel application with gprof. As with the serial case, include the '-pg' flag in the compile and link line. For a parallel program, a single copy of the binary is executed for each task, and gprof will create a unique call graph output for each. In order to be able to distinguish the output for each task, gprof supports output file parametrization. Each output file is parametrized based on the individual task's Unix process ID. This features is enabled by setting the environment variable "GMON_OUT_PREFIX" to a non-null value. After this has been set, each output file will be named "gout.*ProcessIDNUM*" (where ProcessIDNUM changes).

   There is one additional feature that allows the user to aggregate all the data from the tasks to one file in order to make analysis easier. To do this, use the '-s' (sum) option. After the program completes, run "gprof -s gout.*" to create an aggregate output file.

7. **Were there any performance differences between Phase 1 and Phase 2 nodes?**

   As shown in Table 1, there is a performance difference between *Phase 1* and *Phase 2*. *Phase 2* performed significantly faster. We expected this performance difference because the *Phase 2* Haswell Nodes have a performance per core of 41.6 DP FLOPs/sec, whereas *Phase 1* is much slower than this.

1.2 Compiler Flags

"-march=native":	Tells the compiler to that it may use the instructions from ISA (Industry Standard Architecture). On Intel platform "-march=native" along with -O3 (or -ftree-vectorize) will likely generate instruction sets that fully supports AVX registers. Simply put, this flag generates instructions that are more suited for particular architecture.
"-fomit-frame-pointer":	Omits the storing of stack frame pointers during function calls if the function does not require one. Often this is not necessary for simple functions but may be beneficial for complex functions.
"-floop-block":	Creates loop blocks to improve cache hits. Block tile sizes can be specified by using loop-block-tile-size parameter (default is 51 iterations).
"-floop-interchange":	Interchanges the order of nested loops to improve cache hits. Particularly useful when N is larger than the cache line.
"-floop-strip-mine":	Converts a single loop into a nested loop such that it's formed into blocks. Particularly useful for vectorisation and memory access when the strip size is bounded by AVX size and/or cache line.
"-funroll-loops":	Unrolls loops whose number of iterations can be determined at compile time or upon entry to the loop. Improves pipelining, and possibly locality.
"-flto":	Inter-Procedural Optimisation (IPO) based optimisation, which analyses the entire source code for optimisation. This is different to standard compiler optimisation which only analyses single function or single block of code.

Table 3: Compiler optimisation flags for GCC

Further optimisation is possible other than setting the optimisation level (-O⟨level⟩) by using additional optimisation compiler flags. GCC provides approximately 230 compiler flags, and ICC approximately 117. Since the compilers provide a large set of optimisation flags, it is not realistic to test all possible combinations. Therefore, it is important to know the features of these flags, and have at least a rough idea how they would work with each other such that it is suitable for the compute system's architecture before testing them. Table 3 summarises the GCC compiler flags [2] given in the worksheet.

Table 5 summarises the ICC compiler flags [3] given in the worksheet.

Note the compiler flags provided above lie in the scope of architecture optimised instruction generation and loop transformations, which work independently. The loop transformations from the list above do not conflict with each other since they are aimed to perform similar optimisations, but on different memory access patterns. Unlike the old versions, the current version of the LULESH benchmark now supports block-structured domains in order to avoid strip-wise memory access. Furthermore, considering that both Sandy bridge and Haswell processors support vectorisation, loop transformation could be beneficial, as long as they do not conflict with each other, for both architectures. Therefore, the following "guess" of best flag combinations could be considered for GCC compilers on both Phase 1 and Phase 2:

"-xHost":	Tells the compiler to generate instructions for the highest instruction set available on the compilation host processor. Intended to generate optimised instruction sets for Intel microprocessors, and although it may (or may not, depending on the host processor) work on non-Intel microprocessors (summary table available from Intel website), it may or may not optimise the instruction sets to the same degree as it would on Intel microprocessors. This is equivalent to "-march=native" but tailored to Intel processors.
"-unroll":	Sets maximum number of times to unroll loops. This applies only to loops that the compiler determines should be unrolled. Equivalent to "-funroll-loops".
"-ipo":	Inter-Procedural Optimisation. Equivalent to "-flto".

Table 5: Compiler optimisation flags for ICC

- -march=native -fomit-frame-pointer -floop-block -floop-interchange -floop-strip-mine -funroll-loops
- -march=native -fomit-frame-pointer -floop-block -floop-interchange -floop-strip-mine -funroll-loops -flto

However, it is difficult to judge that such a combination will necessary work well. Hence, in order to make sure there are no conflicting flags, test runs were conducted in the following manner:

1. Only include one flag and run each test case (7 test runs)
2. Keep the best performing flag
3. Include the flag(s) which have been kept and one other flag and run each possible test case
4. Repeat steps 2 and 3 until there is no flag which can be added which improves performance

Each test run was conducted five times so that an average value could be used to determine the best performing flag.

Note that for this section on compiler optimisation flags, elapsed time (output from LULESH benchmark) is considered as a performance metric. This would not be appropriate for the following exercises since LULESH benchmark is weak-scaled. However, for a single threaded sequential test runs (hence the domain size is fixed and small) elapsed time is a meaningful performance metric. Table 6 summarises the performance of some of the flag combinations.

Flag Combination	**Speedup**
-floop-interchange	1.13
-floop-block	1.13
-floop-block -flto	0.99
-floop-block -floop-interchange	1.12

Table 6: Speedups from different compiler optimisation flags (GCC-Phase 1) including -march=native -funroll-loops -floop-strip-mine -fomit-frame-pointer (omitted from table due to large size). Speedup here is defined as elapsed time (no flag)/elapsed time (with flags)

Throughout the test runs, "-march=native" gave the best performance improvement. We believe this is because of "-march=native" and -O3 combination which generates an instruction that fully uses vectorisation modules. Same results were obtained on Phase 2 (Haswell). "-flto" on the other hand, degraded the performance significantly. It was also interesting to observe "-floop-block" produce good speedup, considering many of the data structures in LULESH benchmark are one-dimensional arrays. LULESH benchmark now also supports block-wise memory access, and we believe this could be the reason. Same results were obtained on Phase 2 node, but faster elapsed time.

Therefore, the following best flag combination was obtained for GCC on both Phase 1 and Phase 2 nodes.

-march=native -fomit-frame-pointer -floop-block -floop-interchange -floop-strip-mine -funroll-loops

Similarly for ICC, we came up with the following initial "guess" of best flag combination and the test runs were conducted same as for GCC:

- -xHost -unroll
- -xHost -unroll -ipo

Flag Combination	Speedup
-xHost -unroll	1.08
-xHost -ipo	0.99
-march=native -unroll	1.07
-march=native -ipo	0.98
-march=native -xHost	1.11
-march=native -xHost -unroll	1.10

Table 7: Speedups from different compiler optimisation flags (ICC-Phase 1). Again, speedup here is defined as elapsed time (no flag)/elapsed time (with flags)

Table 7 summarises the performance for each combination.

Again, "-xHost" produced the best performance improvement, and "-ipo" deteriorated the performance. In principle, "-xHost" is intended to give a more optimised instruction set for Intel processors than "-march=native". Although this depends on the processor, since we are using Intel processors it would be advisable to use "-xHost" over "-march=native" when using Intel compilers. Again, the same results were obtained on Phase 2 node. Please note that speedup-wise, "-xHost -march=native" produced the best performance (Table 7). However, considering that these two flags are essentially the same, one being more general and the other being tailored to Intel compilers, it would be more advisable to use "-xHost" for ICC.

Therefore, the following best flag combination was obtained for ICC on both Phase 1 and Phase 2 nodes:

-xHost -unroll

Although the questions from the worksheet are answered above throughout the section, the following provides a summary of the answers:

Question 4.2.2.1: icc provides approximately 117 compiler optimisation flags and gcc approximately 230.
Question 4.2.2.2: Since there are too many compiler optimisation flags available, it is not feasible to test every combinations. Hence, it is important to pre-guess the best combination considering both the code (e.g. Data Structures, Loops, Operations, etc), and the compute system's architecture.
Question 4.2.2.3: *-march=native -fomit-frame-pointer -floop-block -floop-interchange -floop-strip-mine -funroll-loops* for gcc, and *-xHost -unroll* for icc. Both applies to Phase 1 and Phase 2 nodes.
Notes on submission files: generateBinary.sh generates multiple makefiles with different compiler optimisation flags, and adjusts job.cmd file accordingly. Flag combinations for next test runs depend on the result, hence both generateBinary.sh and job.cmd files must be adjusted accordingly. For the submission only the scripts for the very first test run (one flag for each Makefile) are provided as examples.

1.3 Optimization Pragmas

Tables 8 and 10 summarise some of the possible #pragma directives in GCC and ICC respectively.

Table 8: #pragma from GCC documentation

#pragma GCC optimize:	Allows to set global optimization options for functions defined later in the source file. *Syntax:* #pragma GCC optimize ("string"...)
#pragma GCC ivdep:	Allows the programmer to assert that there are no loop-carried dependencies which would prevent SIMD instructions. *Syntax:* #pragma GCC ivdep

1.3.1 #pragma from directives in Intel's documentation:

1.3.2 Questions

1. Difference between simd, vector, and ivdep:

#pragma ivdep instructs the compiler to ignore assumed vector dependencies, meaning that if the compiler sees a possibile dependency, this assumption will be ignored. However, a proven dependency will not be ignored by the compiler using this command. Using **#pragma ivdep** would not automatically vectorize the loop but it can be used along with **#pragma vector** in order to indicate to the compiler that the loop should be vectorized.

#pragma vector indicates to the compiler that the loop should be vectorized. **#pragma vector** and can have several arguments that indicate how the loop should be vectorized. For instance, if the argument 'aligned' is used, then the compiler is instructed to align data movement instructions for all array references when vectorizing. The compiler will vectorize regardless of normal heuristic decisions and profitability, but will only vectorize when it is legal to do so. Furthermore, nested loops require a preceding **pragma** statement for each for loop.
In order to make the compiler ignore all vector dependencies, assumed or proven, and enforce vectorization, **#pragma simd** must be used. If the compiler is unable to vectorize a loop, a warning will be emitted.

Table 10: #pragma from Intel documentation

#pragma simd	Enforces vectorization of loops, regardless of dependency assumptions the compiler may have. This directive can lead to unintentional calculations if not careful.
#pragma ivdep	Instructs the compiler to ignore assumed vector dependencies. This directive overrides the compiler's assumed vector dependencies, and can be used to complement **#pragma vector**. This command is safe to use, and still allows the compiler to decide whether or not to vectorize a loop based on profitability.
#pragma loop_count(n)	Specifies the iterations for a for loop. Syntax: #pragma loop_count(n). There are also other arguments that can be used with this #pragma such as min() max(), and avg().
#pragma inline	The inline pragma is a hint to the compiler that the user prefers that the calls in question be inlined, but expects the compiler not to inline them if its heuristics determine that the inlining would be overly aggressive and might slow down the compilation of the source code excessively, create too large of an executable, or degrade performance.
#pragma forceinline	The forceinline pragma indicates that the calls in question should be inlined whenever the compiler is capable of inlining.
#pragma noinline	The noinline pragma indicates that the calls in question should not be inlined.
#pragma unroll and **#pragma nounroll**	Indicates to the compiler to unroll (or to not unroll) a counted loop. **n** is the unrolling factor representing the number of times to unroll a loop; it must be an integer from 0 through 255.
#pragma distribute_point	This directive suggests to the compiler to split larger loops in to smaller ones.
#pragma unroll_and_jam	This directive will partially unroll one or more loops higher in the loop nest than the innermost loop. It then jams the loops back together to allow more reuses in the loop. Ensure that this command is not done on the innermost loop, as it is not effective on innermost loops. If this pragma is specified, the compiler will make sure to unroll and jam, if legal to do so. **#pragma unroll_and_jam** must precede the for loop it affects and the variable **n** will specify the amount the loop should be unrolled.
#pragma nofusion	prevents a loop from fusing with adjacent loops. Lets you fine tune your program on a loop-by-loop basis. It is placed before the loop that should not be fused.

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

In order to decide where to add a #pragma directive within the source code, we analyzed the gprof.out file. From both *Phase 1* and *Phase 2* gprof.out files, the function which took the most calculation time was **CalcHourglassControlForElems**. After analysing the function, we found the following piece of code that could lead to a possible loop dependency assumption:

```
        for(Index_t ii=0;ii<8;++ii){
        Index_t jj=8*i+ii;
```

After a simple check, it became clear that the the following is true: $jj \geq ii$ and therefore the loops are independent. We chose to use **#pragma ivdep** to optimize the code for a few reasons. If **#pragma vector** had solely been used, the compiler may still not have vectorized the loop because it assumes a vector dependency. We also considered using **#pragma simd**, but decided that **#pragma simd** should only be used in critical situations, due to the fact that the simd directive tells the compiler to ignore all assumed or proven dependencies. We could have also used **#pragma vector** complimented with **#pragma ivdep**, but decided against this because then the compiler would vectorize regardless of normal heuristic decisions and profitability. By using **#pragma ivdep**, the compiler ignores the assumed dependency but is given the freedom to decide whether or not vectorizing the following loop would be profitable.

We also tested the performance of the three stated #pragma directives (See (Table 11)) and found the speedup to be very similar. This further confirmed the decision to use **#pragma ivdep**. This directive arguably provides the same performance increase along with it being safe coding practice.

Here is our implementation of **#pragma ivdep**.

```
#pragma ivdep
for(Index_tii=0;ii<8;++ii){
        Index_t jj=8*i+ii;
        ...
}
```

The following table (Table 11) depicts the speedups gained from each pragma. ICC compilers were used for these test runs on Phase 1 node. The speedup here is defined as time(no pragma)/time(with pragma). Each binary execution was repeated five times. Grind time (overall) was used as the performance metric since LULESH benchmark is a weak-scaled benchmark. More detailed explanations on the choice of performance metric is provided in the following sections. The test runs were run on Phase 1 node using ICC with the compiler optimisation flag combination mentioned on Section 1.2

Pragma	Speedup
#pragma ivdep	0.89
#pragma vector	0.94
#pragma simd	0.96

Table 11: Speedups obtained from each OpenMP pragmas. Grind time (overall) was considered as the performance metric.

1.4 Inline Assembler

1.4.1 Comparison between AT&T and Intel inline assembler code:

```
//AT&T:
asm("movl %ecx, %eax");

//Intel:
mov eax, [ecx];
```

1.4.2 Questions

What is inline assembler?

Inline assembler is a feature provided by some compilers that allows for assembly language to be directly coded within high-level language (C/C++ and others). Inline assembler is mostly used for extreme optimization needs or for code that must perform a highly hardware-specific function. Also, inline assembler can be used for direct facility to make system calls.

Is inline assembler necessarily faster than compiler generated code?

Handmade inline assembler code theoretically should always be as fast or faster than compiler generated code, but this comparison is heavily dependent on the programmers programming skill and what specifically said programmer is coding in inline assembler code. The compiler is able to create highly efficient and optimized assembly level code for almost all cases. The compiler assembly code will usually be more optimized than handmade assembly code made by a regular programmer. Inline assembler code should only be used for

few and popular performance critical routines or to expose a hardware-specific function that the high level code may not be able to expose.

On the release of a CPU with new instructions, can you use inline assembler to take advantage of these instructions if the compiler does not support them yet?

asm is able to do hardware-specific functions and to make direct system calls. That would seem to suggest that it should be able to take advantage of new instructions even if the compiler doesn't support them yet.

What is AVX-512?

AVX-512 is a 512-bit extension to the previous 256-bit Intel Advanced Vector Extensions 2 (AVX2). AVX's in general are extensions to the x86 instruction set architecture and each AVX update usually increases SIMD ability. AVX-512 provides better vector performance than the previous AVX2 due to double width registers and 2x more registers than previously (512 compared to 256). Furthermore, AVX-512 introduces new instructions to accelerate specific tasks for modern workloads.

Which CPUs support it? AVX-512 can only be used on Intel Xeon Scalable Processors or Intel Xeon Phi Processor Product Family. More specifically:

- Intel Xeon Phi x200 (Knights Landing)
- Intel Xeon Phi Knights Mill
- Intel Skylake-SP,
- Intel Skylake-X
- Intel Cannonlake
- Intel Ice Lake

Note: Intel Xeon Gold and Xeon Platinum editions include additional performance enhancements when using AVX-512.

Is there any compiler or language support for these instructions at this moment?

There is in fact compiler and language support for the instructions, most Intel compilers, libraries, and analysis tools, currently have or will be updated in the near future to support AVX-512. For instance, the Intel C++ Compiler (icpc) has AVX-512 support. Use the **-xMIC-AVX512** flag to request that the compiler use the Intel AVX-512 ISA to generate executable code.

2 Performance Scaling

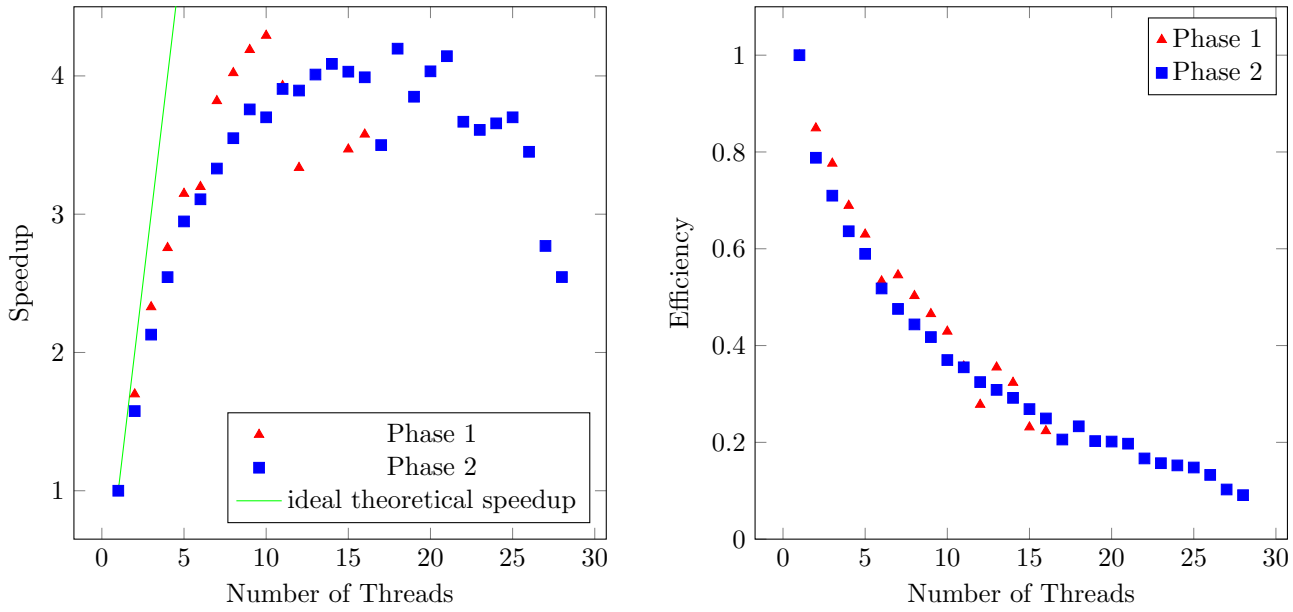
According to the LULESH documentation[1] the performance of LULESH can be expressed using either the wall clock time or the grind time. The wall clock time gives how long a version takes to perform the simulation. Grind time gives the time it takes to update a single zone at each time step. Therefore for a fixed number of zones the wall clock time is directly proportional to the grind time. However, the LULESH benchmark is weak scaling. This means that when the number of processes is increased the domain also increases proportionally. As a result the wall clock time cannot be used to judge the performance improvements. However, the size of a zone remains fixed so the overall grind time can be used as a performance metric. This is what will be used to construct the figures in the following sections.

The figures presented in the following sections show the speedup and efficiency of the LULESH program on the phase 1 and phase 2 nodes of Supermuc. The speedup and efficiency are defined as follows:

$$speedup = \frac{P_p}{P_1} \tag{1}$$

$$efficiency = \frac{P_p}{P_1 n_t} \tag{2}$$

where P_p is the performance on p nodes (defined previously as the overall grind time), and n_t is the number of threads multiplied by the number of processes (we assume 1 thread when OpenMP is not used and 1 process when MPI is not used).



(a) Speedup (b) Efficiency

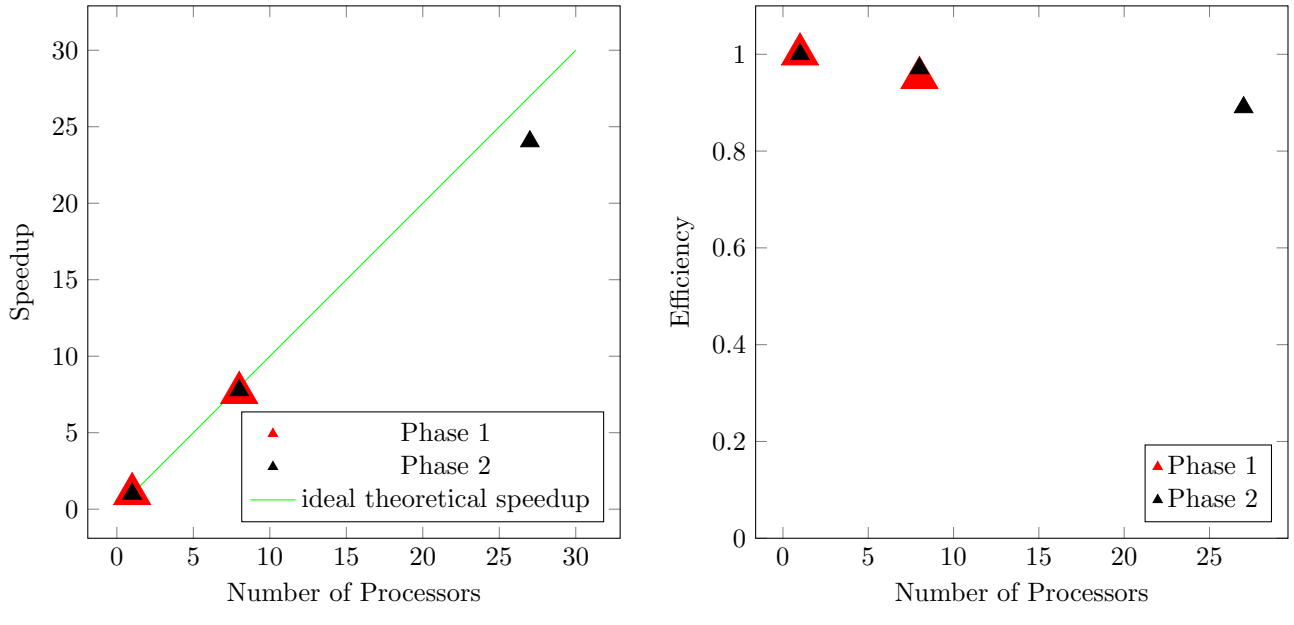
Figure 1: Performance of LULESH with OpenMP

2.1 OpenMP

The only restriction on the possible number of threads used by the program is that this number should not exceed the number of cores on the node, 18 for phase 1 nodes and 28 for phase 2 nodes. This means that the scalability of the benchmark can be easily deduced from figure 1a. The scalability is approximately linear for the first 11 threads on both phase 1 and phase 2 nodes. Once more threads were used the scalability stopped being even approximately linear and the parallel overhead began to become significant. The maximum performance was seen at 13 threads on the phase 1 node and at 18 threads on the phase 2 node. After this point the parallel overhead dominated and reduced the speed of the program.

2.2 MPI

The LULESH benchmark is constructed in such a way that the number of processes must be the cube of an integer. In addition the number of processes cannot exceed the total number of cores available. In the phase 1 nodes there are 16 cores per node. This means that the benchmark can only be run with 1 or 8 processes.



(a) Speedup (b) Efficiency

Figure 2: Performance of LULESH with MPI

In the phase 2 nodes there are 28 cores per node. This means that one extra test can be run. Thus the benchmark can be run on a phase 2 node with 1, 8 or 27 processes.

The results of these tests can be seen in figure 2.

It is unfortunate that it is not possible to run more tests. It is of course always possible to fit a linear relation to two points, however as the results for phase 1 and phase 2 are so similar it is reasonable to assume that the scalability is similar. The scalability of phase 2 appears to be linear, this is confirmed by the proximity of the results to the ideal theoretical results. The maximum performance was seen with the maximum number of processes, 8 for phase 1 and 27 for phase 2.

This is in contrast with OpenMP where the scalability levelled off when a large number of threads were used. This may be because LULESH is weak scaling with respect to MPI but not with respect to OpenMP. This means that when the number of processes increases, the problem size increases proportionally, however this is not the case when the number of threads increases. When more processes or more threads are added, the parallel overhead increases, however if the problem size also increases then the overhead is a smaller percentage of the overall computation, thus its effect is less significant. This explains why the overhead does not eventually dominate for MPI as it does for OpenMP.

2.3 MPI + OpenMP

Combining OpenMP and MPI means that the previously described restrictions on the number of processes and threads must still be respected, however in addition they must be combined. Therefore the product of the number of threads with the number of processes must be less than the number of cores on the node. The resulting possible combinations can be seen in table 12.

Number of Processes	Possible number of threads on Phase 1	Possible number of threads on Phase 2
1	1-16	1-28
8	1, 2	1, 2, 3
27	not possible	1

Table 12: Possible combinations of numbers of processes and threads on Phase 1 or Phase 2 nodes of Supermuc

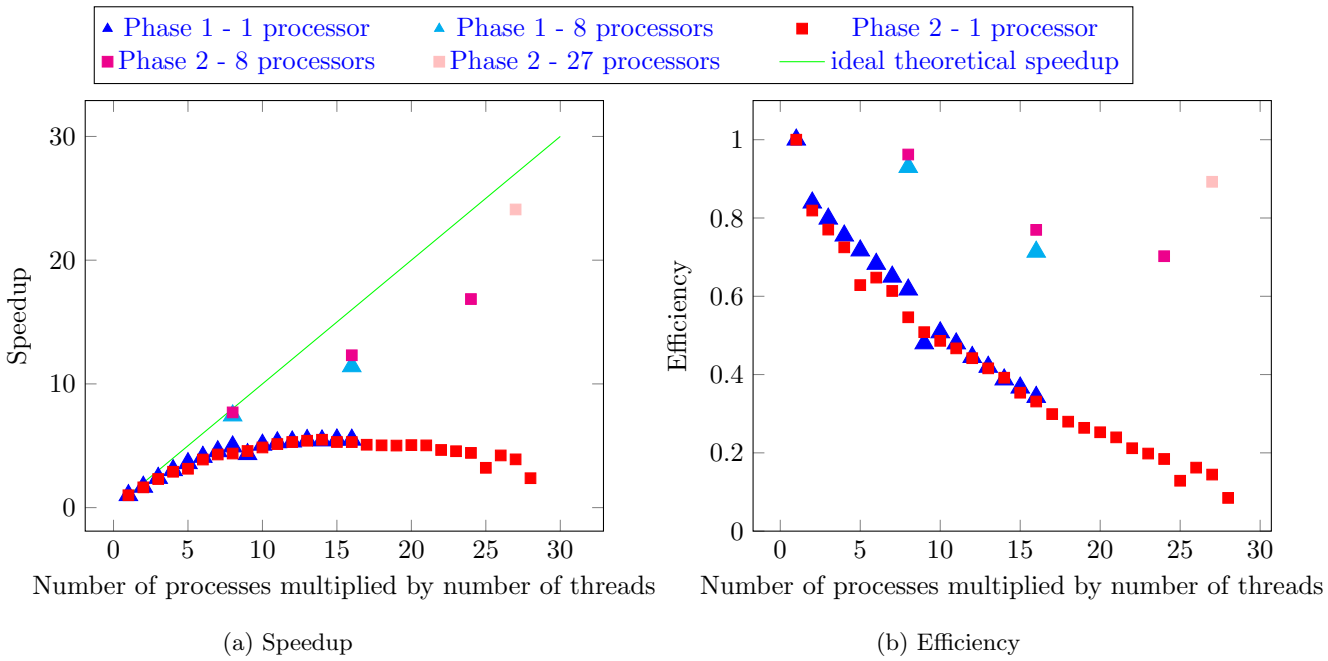
The results of these tests can be seen in figure 3. We see that the more processes are used, the better the speed-up. In contrast an increase in the number of threads does not necessarily lead to an increase in speed-up. Small numbers of threads improve the performance, but large numbers lead to a large amount of parallel overhead which prevents speed-up. The best performance on the phase 1 node was seen using 8 processes and 2 threads. The best performance on the phase 2 node was seen using 27 processes and 1 thread. These results were also the overall best results

These results are similar to those measured in sections 2.1 and 2.2 in that the performance scales linearly in the number of processes and scales linearly at first in the number of threads, with parallel overhead dominating when large numbers of threads are used. This is what we expected as most parallel applications follow a similar speedup curve. This curve begins with a quasi-linear region as the speedup increases with the number of processes/threads then levels off and sometimes even deteriorates as the parallel overhead dominates. This would imply that a reasonable number of threads (a large amount but not the maximum amount) would produce the optimal set-up. This logic however only applies to OpenMP as LULESH is weak scaling with respect to MPI. Thus the overhead increases with the problem size. This means that the levelling off of the curve would appear much later if at all. Indeed with the number of processes available to us, we were not able to see this levelling off.

2.4 Questions

Although the questions from the worksheet are answered above throughout the section, the following provides a summary of the answers:

Question 5.1.2.1: Linear scalability was achieved for the first 10 threads after which the parallel overhead dominated and reduced the speed of the program.
Question 5.1.2.2: The maximum performance was achieved with 13 threads on the phase 1 node and 18 threads on the phase 2 node. The result was therefore not the same on the different node types.



(a) Speedup

(b) Efficiency

Figure 3: Performance of LULESH with OpenMP and MPI

Question 5.2.2.1: The LULESH benchmark is constructed in such a way that the number of processes must be the cube of an integer. As a result, the benchmark can only be run with 1 or 8 processes on a phase 1 node, and with 1, 8 or 27 processes on a phase 2 node.
Question 5.2.2.2: Linear scalability was achieved.
Question 5.2.2.3: The maximum performance was achieved with 8 processes on the phase 1 node and 27 processes on the phase 2 node. The result was therefore not technically the same on the different node types, however the best performance was achieved with the maximum number of processes on both node types.
Question 5.2.2.4: The results had a much better linear scaling than OpenMP where the scalability levelled off when a large number of threads were used. This may be because LULESH is weak scaling with respect to MPI but not with respect to OpenMP. This means that when the number of processes increases, the problem size increases proportionally, however this is not the case when the number of threads increases. When more processes or more threads are added, the parallel overhead increases, however if the problem size also increases then the overhead is a smaller percentage of the overall computation, thus its effect is less significant. This explains why the overhead does not dominate for MPI as it does for OpenMP.
Question 5.3.2.1: The valid combinations of threads and processes can be found in table 12
Question 5.3.2.2: The best performance on the phase 1 node was seen using 8 processes and 2 threads. The best performance on the phase 2 node was seen using 27 processes and 1 thread.
Question 5.3.2.3: The maximum performance was achieved with 8 processes and 2 threads on the phase 1 node and 27 processes and 1 thread on the phase 2 node. The result was therefore not technically the same on the different node types, however the best performance was achieved with the maximum number of processes, and the maximum number of threads available as a result, on both node types.
Question 5.3.2.4: These results are similar to those measured in sections 2.1 and 2.2 in that the performance scales linearly in the number of processes and scales linearly at first in the number of threads, with parallel overhead dominating when large numbers of threads are used.
Question 5.3.2.5: The fastest solution on the phase 1 node was OpenMP/MPI with 8 processes and 2 threads. The fastest solution on the phase 2 node was OpenMP/MPI with 27 processes and 1 thread. Although the difference between this result and the result achieved using only MPI with 27 processes is negligible so it seems likely that both solutions carry out the same operations, ignoring OpenMP as there is only 1 thread.
Question 5.3.2.6: This is what we expected as most parallel applications follow a similar speedup curve. This curve begins with a quasi-linear region as the speedup increases with the number of processes/threads then levels off and sometimes even deteriorates as the parallel overhead dominates. This would imply that a reasonable number of threads (a large amount but not the maximum amount)

would produce the optimal set-up. This logic however only applies to OpenMP as LULESH is weak scaling with respect to MPI. Thus the overhead increases with the problem size. This means that the levelling off of the curve would appear much later if at all. Indeed with the number of processes available to us, we were not able to see this levelling off.

References

[1] LULESH Programming Model and Performance Ports Overview
https://codesign.llnl.gov/pdfs/lulesh_Ports.pdf

[2] Using the GNU Compiler Collection (GCC): Optimize Options
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[3] User and Reference Guide for the Intel® C++ Compiler 15.0
https://software.intel.com/en-us/compiler_15.0_ug_c

[4] Intel Developer Zone
https://software.intel.com/en-us/node