Assignment 1: Single Node Performance
Programming of Super Computers

Group 10
Anibal Siguenza Torres
Santiago Narvaez Rivas
Gajendra Gulgulia

November 20, 2017

4 Performance Baseline

Questions 4.1

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

For both Phase 1 and Phase 2 of SuperMUC, it was observed that the following functions took 80% or more of the time : **main. CalcHourglassControlForElems, CalcKinematicsForElems, CalcQForElems**.

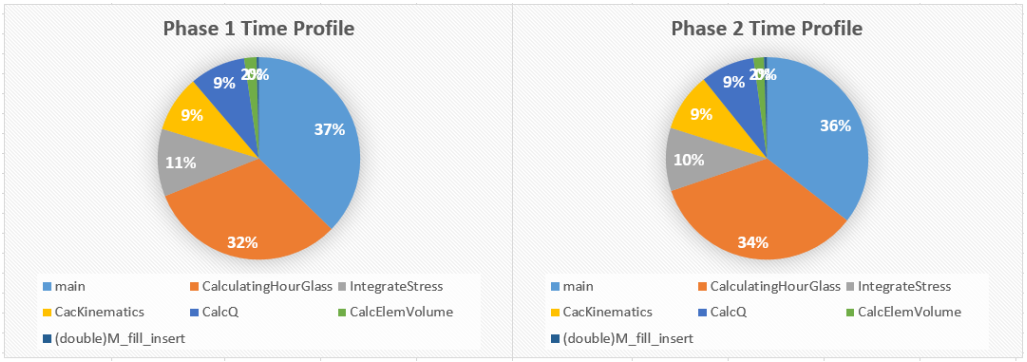

Figure 1: Plot representing the percentage of time spent on functions

Is the measured execution time of the application affected by gprof?

It was observed with the benchmark run, that indeed the execution time is slightly affected by *gprof*

Table 1: Comparison timing for run times with and without gprof flgs (on phase 1)

Sl No	Time Parameter	w/o gprof flag	with gprof flag
1	real	49.244	50.373
2	user	46.699	47.271
3	sys	2.42	3.064
4	elapsed	49.21	50.35
5	grind	1.9554	2.0009
6	FOM	511.38	499.77353

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

Explicitly gprof cannot analyze the loops but if the loops are wrapped within a function call we can get how much time was spent on each loop.

Is gprof adequate for analysis of long running programs?

If the program is long running and we are only interested in profiling, then grpof seems a suitable tool because gprof relies on frequency of sample collection, which by default is 100 Hz. If the running time of gprof is very small and if within this small duration it calls many functions, then it is highly likely that the samples size for profiling are very less and hence the samples of profiled functions are not reliable. Otherwise the instrumentation overhead for profiling can increase the run time of a program and the total execution time may not reflect the actual runtime of an application, which can simply be measured by the calling the suitable time functions provided by operating system kernel.

Is gprof capable of analyzing parallel applications?

No gprof is not good in analyzing parallel applications, i.e it is neither designed to profile multi-threaded applications nor distributed memory parallel applications.

What is necessary to analyze parallel applications ?

The problem of analyzing parallel applications can be divided into two parts:

1. **Performance Analysis** : This tries to filter the huge volume and statistics about a parallel program's execution and tries to generate useful information for the user
2. **Instrumentation** : This focuses on how to efficiently collect enough useful run time information about the application's execution as a side effect of its execution.

. (source: //www.cs.umd.edu/ hollings/papers/progEnv.pdf)

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

From plot 2 it can be concluded that phase 2 is faster than phase 1. This can be because of the presence of two 128 bit FMUL or fused multiplication addition registers in Hasewell processors.

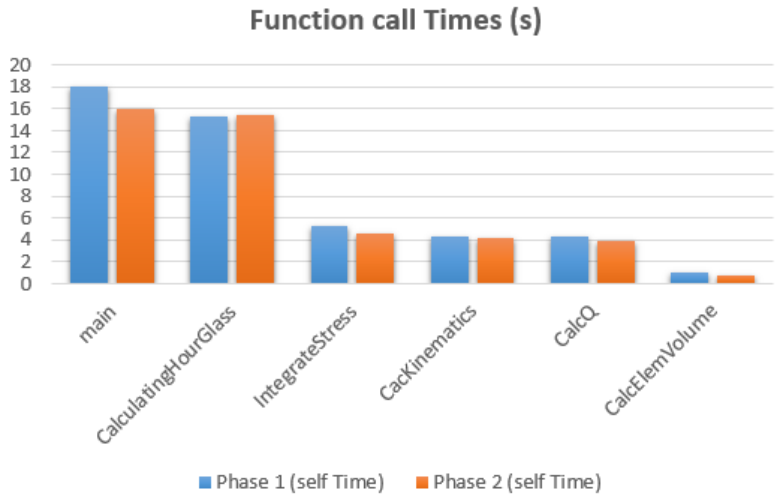

Figure 2: Absolute self run times of functions in phase1 and phase2 of superMUC

Questions 4.2.2

How many optimization flags are available for each compiler (approximately)?

For GCC, it is possible to use the command "gcc –help=optimizers" to check all the option control optimization. Assuming that the output shows one flag per line (which is true for almost all of them), one can execute "gcc –help=optimizers | wc -l" to approximate count the total number of options. For GCC 6, the result was around 220 different flags.

For icc it is harder to estimate the total number of flags. The optimization section displayed after run "icc –help" only shows 18 flags. Nevertheless, there are many flags that are used for optimization that are classified in other sections (i.e., the "–march=<cpu> flag is in the Code generation section).

would it be realistic to test all possible combinations of available compiler flags in these compilers?

No. Given the total number of optimization flags, it would make no sense to test all the possible flag combinations. In principle, one should have an idea of which flags might improve the performance of the code, and then make tests only with a subset of the optimization options.

Which compiler and optimization flags combination produced the fastest binary?

Table 2: Flag combinations with fastest binaries

	flag	**time (s)**
gcc phase 1	-march=native -fomit-frame-pointer -floop-block -floop-strip-mine -funroll-loops	40.74
gcc phase 2	-march=native -fomit-frame-pointer -floop-block -floop-strip-mine	35.28
icc phase 1	-march=native -unroll	37.42
icc phase 2	-march=native -xHost	33.01

In Table 2 is showed the flags combination that produced the fastest binaries for each compiler and each phase.

Given that the flag -fomit-frame-pointer is incompatible with -pg, and therefore there was no way to profile the code, for the subsequent GCC pragma optimization the second fastest flag combinations were used:

- GCC GCC phase 1: -march=native -floop-block -floop-strip-mine -funroll-loops (40.88 s)
- GCC phase 2: -march=native floop-interchange (35.4 s)

As it can be seen, the difference in runtime is not significant, but using these flags it is possible to profile the code and optimize the most relevant functions.

Questions 4.3.2

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

1. **#pragma simd** : This pragma guides the compiler to do more vectorization, i.e it assists the compiler, while still maintaining the rules that compilers would normally apply to the automatic optimizations and vectorization.

2. **#pragma vector** : This pragma tells the complier to vectorize the loops when it is legal to do so by ignoring the normal compiler optimization heuristics. In a nested loop, the compiler applies pragma only to the outer loop and each of the inner loop needs a separate pragma statement.

   Comparing this to **#pragma simd**, the **#pragma vector** will force the compiler to vectorize whenever it is legal, whereas in **#pragma simd** the compiler is still responsible for checking the profitability of application of the pragma. [1] [2]

[1] see: https://software.intel.com/en-us/node/524555
[2] see: https://software.intel.com/en-us/node/524559

3. #**pragma ivdep** : In general Intel's compiler treats assumed dependencies as an extreme case of proven dependencies [3], i.e, in in simple words, if the compiler suspects that there might be a dependency, it will assume that the dependency actually exist and prevents vectorization in the loop. #**pragma ivdep** overrides this behaviour and instructs the compiler to ignore the assumed dependencies. In this case, the programmer has to be sure that the assumed dependencies are safe to be ignored.

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

Tables 3 and 4 show the optimal pragmas for intel and gnu compilers used in both phase 1 and phase 2 nodes.

Table 3: optimal GNU (gcc) and Intel(icc) pragmas for Phase1 node

Phase 1		
compiler	icc	gcc

#pragma	#pragma loop_count (64)	#pragma GCC Optimize ("O3")
		__attribute__((no_return))

Function on which #pragma was applied	EvalEOSForElems( )	ApplyMaterialPropertiesForElems( )*

Table 4: optimal GNU (gcc) and Intel(icc) pragmas for Phase2 node

Phase 2		
compiler	icc	gcc

#pragma	#pragma loop_count(20)	#pragma GCC Optimize ("O3")
		__attribute__((always_aligned))

Function on which #pragma was applied	EvalEOSForElems( )	ApplyMaterialPropertiesForElems( )

The reason for using the particular location is because after profiling the code that ran the fastest with compiler flags as mentioned in table 2, the execution was profiled with the corresponding combination of the compiler flag for intel/gnu compiler in both phase 1 and phase 2 for checking the function call that took the longest to execute. We therefore applied the pragmas to the function that took the largest amount time with the corresponding compiler flags.

In the case of intel compiler (for both phase 1 and 2), the pragma **loop_count(n)** defines the number of times a for loop can be executed [4], and thus this pragma is applied at each for loop inside the correct function (EvalEOSForElems in this case).

In the case of gnu compiler (for both phase 1 and 2) , needs a special declaration [5] as shown below :

```
//example to demonstrate the usage of #pragma GCC optimize
.
.
//function declaration
#pragma GCC optimize ("O3")
void foo() __atribute__((attribute_type));
.
.
.
//function definition
```

[3] see: https://software.intel.com/en-us/node/524501
[4] See: https://software.intel.com/en-us/node/524502
[5] See: https://gcc.gnu.org/onlinedocs/gcc-5.3.0/gcc/Function-Attributes.html#Function-Attributes

```
void foo(){
//Do something
    //within the function

}
```

This semantics is followed and applied to the code at d line number 177 in the lulesh.cc file to the correct function (ApplyMaterialProperitesForElems in this case).

Questions 4.4.2

Is inline assembler necessarily faster than compiler generated code?

Not necessarily. A good compiler will generate highly optimized code, and match the performance of such program by writing directly the assembly code is usually a hard task. Now, there are situations in which the programmer might choose to write the assembly for short, highly used and performance critical routines. In those cases the directly written assembly code is expected to be faster.

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?

Yes. According to the GCC 7 documentation[6] "GCC does not parse the assembler instructions themselves and does not know what they mean or even whether they are valid assembler input". This basically means that you can use new assembly instructions which are not yet supported by the compiler. The documentation for GCC 4 goes even further, adding that "The extended asm feature is most often used for machine instructions the compiler itself does not know exist"[7].

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

AVX-512 stands for Advanced Vector Extensions 512. AVX-512 allow the use of 512-bit SIMD instruction. This basically mean that "programs can pack eight double precision or sixteen single precision floating point numbers , or eight 64-bit integers, or sixteen 32-bit integers within the 512-bit vectors"[8].

Currently, AVX-512 instructions are supported by Intel® Xeon Phi Knights Landing, Skylake-SP and Skylake-X processors. But one must be careful because only a subset of the instructions might be supported. For example, the Xeon Phi Knights Landing processors support AVX-512F (foundation), AVX-512CD (conflict detection), AVX-512ER (exponential and reciprocal) and AVX-512PF (prefetch), but not AVX-512VL (Vector Length Extensions) among others.

These new instructions are supported by the Intel compiler. To generate Intel AVX-512 instructions for the Intel Xeon Phi processor x200, one should use the option –xMIC-AVX512[9]. GCC, since its 4.9 version, also supports AVX-512 instructions. For example, it is possible to use the flag -mavx512f to use AVX-512F[10].

[6] See: https://gcc.gnu.org/onlinedocs/gcc-7.2.0/gcc/Extended-Asm.html
[7] See: https://gcc.gnu.org/onlinedocs/gcc-4.3.5/gcc/Extended-Asm.html
[8] See: https://software.intel.com/en-us/blogs/2013/avx-512-instructions
[9] See: https://software.intel.com/en-us/articles/compiling-for-the-intel-xeon-phi-processor-and-the-intel-avx-512-isa
[10] See: https://gcc.gnu.org/gcc-4.9/changes.html

5 Performance Scaling

Questions 5.1.1

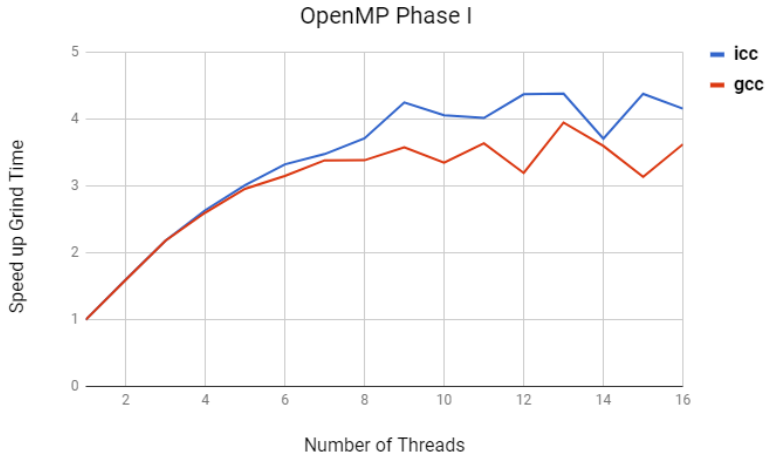

Figure 3: Speed up of OpenMP with icc and gcc compiler on superMUC phase 1

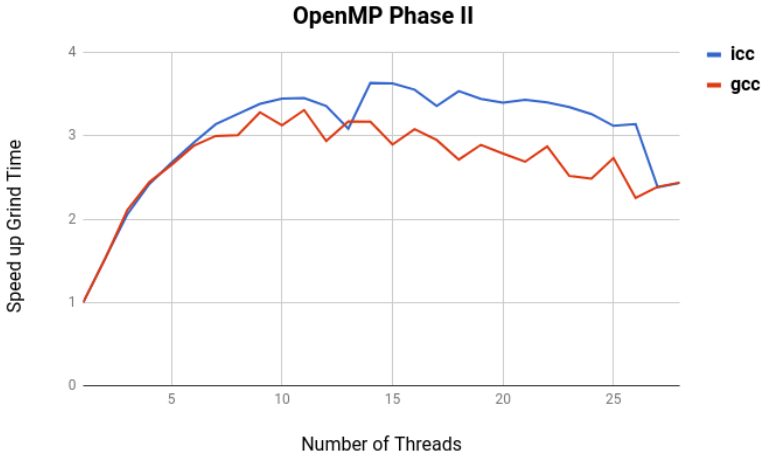

Figure 4: Speed up of OpenMP with icc and gcc compiler on superMUC phase 2

Questions 5.1.2

Was linear scalability achieved?

No, as expected the overhead added to the parallelization process made the speed up less efficient than the number of threads. Also it can be observed a asymptotic behavior as we approach to the max number of threads which is 16. All this can be observed in figures 3 and 4.

On which thread-count was the maximum performance achieved? Was it the same for both types of nodes?

The maximal performance for the phase 1 was achieved with 13 threads for both compilers. In the phase 2 it was 11 threads with gcc compiler and 14 with icc. So it was not the same. Also we can notice that after 9 threads the solutions start to oscillate and start to get flat, so the improvements after this thread is marginal.

Questions 5.2.1

	Phase 1		Phase 2		
Number of Processes	1	8	1	8	27
Grind Time	1.3261553	0.17769951	1.3289695	0.17319654	0.056364756
Speed up Grind Time	1	7.462909155	1	7.673187351	23.57802276

Table 5: MPI benchmark results

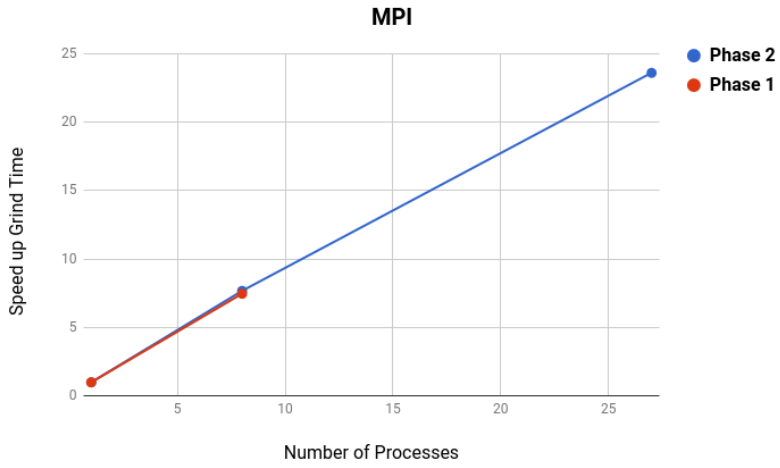

Figure 5: Speed up of MPI on both superMUC phases

Questions 5.2.2

What are the valid combinations of processes allowed?

The test has to be done with one node. Therefore the number of processes can not exceed the number of cores. Phase 1 has 16 cores per node, and Phase 2 has 28 cores per node. Considering that the LULESH benchmark can only take cube numbers (i.e., 1, 8 , 27, etc.) the only values allowed for Phase 1 would be 1 and 8, while for Phase 2 it is valid to have 1, 8 and 27 processes.

Was linear scalability achieved?

It was almost linear for both phases. For the 8 number of processes the efficiency was 93.2% for phase 1 and 95.9% for phase 2. And for the 27 processes on phase 2 the efficiency was 87.3%. So it was not perfect linear, but the scalability was quite efficient.

On which process-count was the maximum performance achieved? Was this the same for both types of nodes?

The maximum performance was with the maximum possible MPI parallelization in both cases. On phase 1 it was with 8 processes giving a grind time of 0.178 and. On phase 2 with 27 processors it was 0.0564. This is extremely different since phase 2 nodes have 28 cores. Allowing a parallelization far beyond the possible in a single phase 1 node. So probably it is not a fair comparison. On the other hand with 8 processes on both phases, phase 2 performs slightly better, but in general the performances are quite similar.

How does the performance compare to the results achieved with OpenMP in section 5.1?

The results of performance with MPI are far better than those with OpenMP. In figure 8 there is a summary of the fastest runs of each section. And the parallelization with MPI using 8 processes is

almost 2 times faster as the best OpenMP with 15 threads on phase 1. On phase 2 this difference is even greater.

Also it is interesting to notice that when OpenMP approximates to the maximum threads the architecture can handle, increasing the threads make only a small difference in the performance, while with MPI the results are really close to be linear even with 27 processes which is 1 less than the maximum processes the node achieves.

Questions 5.3.1

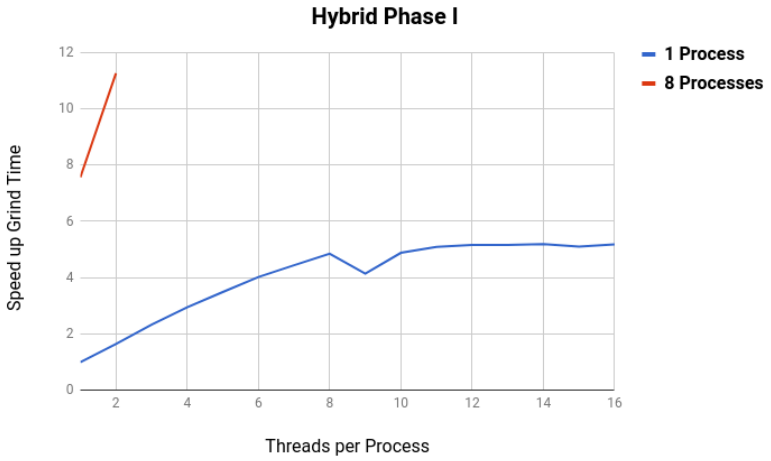

Figure 6: Speed up of MPI with OpenMP on superMUC phase 1

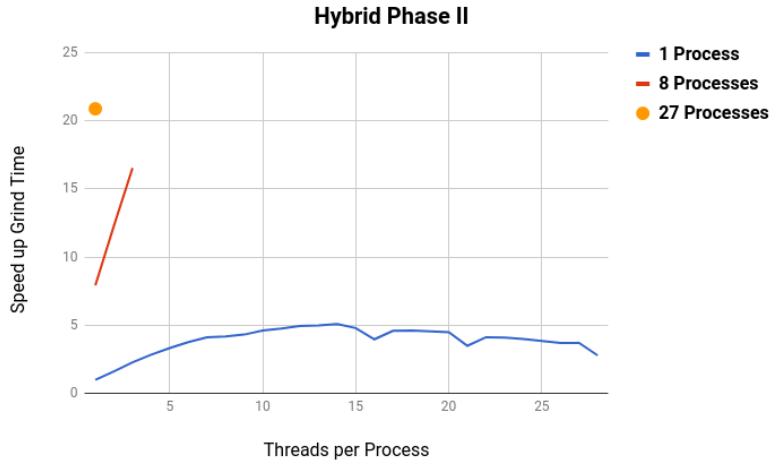

Figure 7: Speed up of MPI with OpenMP on superMUC phase 1

Questions 5.3.2

What are the valid combinations of processes and threads?

The multiplication of the number of processes times the number of threads must not exceed the number of cores per node. Phase 1 has maximum of 16 cores per node, while Phase 2 has a maximum of 28 cores in a node. This means that for Phase 1 one can have 1 or 8 processes with up to 16 or 2 threads respectively. In Phase 2 can have 1, 8 or 27 processes with up to 28, 3 or 1 threads respectively.

Was linear scalability achieved?

No as usual the overhead made that the total number of simulations process ($NumProcesses \times NumThreads$) was never achieved as speed up.

On which combination of processes and threads was the maximum performance achieved? Was this the same for both types of nodes?

On phase 1 the best combination was 8 processes with 2 threads, and in the phase 2 the best combination was 27 processes with 1 thread. It is not the same because of the higher number of processors in phase 2 compared to phase 1.

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

In table 6 we can see the results summarized. With the hybrid implementation the phase 1 was able to improve the results from pure MPI since the resources of the processor was still unused, and introducing OpenMP helps to fill this waisted gap. But in the phase 2 the MPI and Hybrid is practically the same parallelization since both have 27 processes and just 1 thread. Also they have very similar times with a marginal advantage to pure MPI.

		Processes	Num threads	GrindTime
Phase 1	ICC OpenMP	1	15	0.32846916
	ICC MPI	8	1	0.17769951
	ICC Hybrid	8	2	0.12871108
		Processes	Num threads	GrindTime
Phase 2	ICC OpenMP	1	14	0.40281557
	ICC MPI	27	1	0.056364756
	ICC Hybrid	27	1	0.059393674

Table 6: Table with the best performance times of scaling

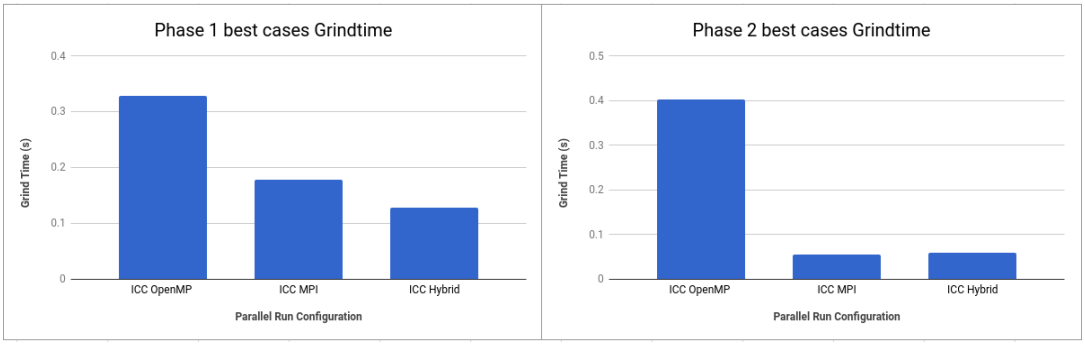

Figure 8: Grindtime comparison of the best cases. The configuration can be seen in table 6

Which solution is overall the fastest for each type of node?

- For the phase 1 it is the Hybrid MPI OpenMP implementation with 8 processors and threads.
- For the phase 2 it is the pure MPI with 27 processors and 1 thread.

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

We would not have guessed in the beginning that MPI would make such a big difference compared to openMP. But definitely now we can say that MPI can make the biggest and more scalable jump, so every-time it is possible to increase the number of processes it is a huge difference. While OpenMP can help to fill the processor capacity gaps, but it does not make as big impact in performance from upon certain number of threads.