Assignment 1
Single Node Performance

Smith Agarwal
Shyam Arumugaswamy
Siddhesh Kandarkar
Ulugbek Kodirov

November 15, 2018

4 Performance Baseline

4.1 GNU Profiler

4.1.1 Required Submission Files

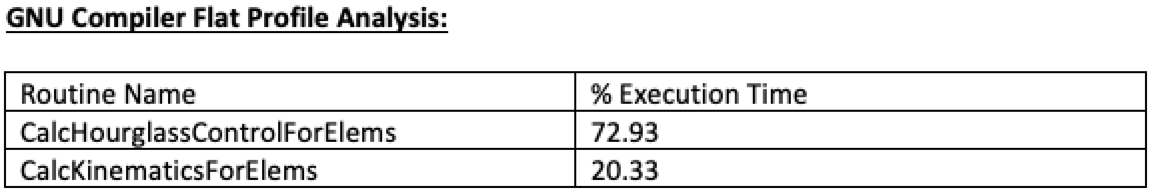
The following gprof.out files can be found in the submission folder for GNU and Intel compilers

1. assignment1.gprof.out.gnu
2. assignment1.gprof.out.intel

4.1.2 Questions

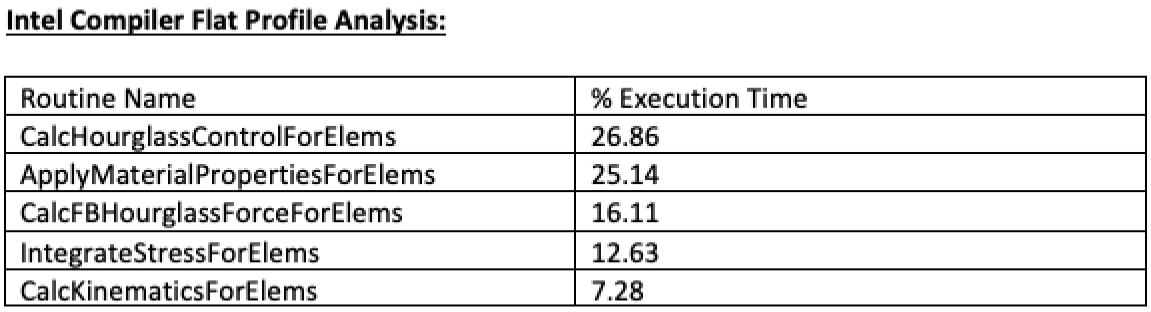
1. **Which routines took 80% or more of the execution time of the benchmark?**

   The following functions calls took approximately 80% of the execution time:



GNU Compiler Flat Profile Analysis:

Routine Name	% Execution Time
CalcHourglassControlForElems	72.93
CalcKinematicsForElems	20.33



Intel Compiler Flat Profile Analysis:

Routine Name	% Execution Time
CalcHourglassControlForElems	26.86
ApplyMaterialPropertiesForElems	25.14
CalcFBHourglassForceForElems	16.11
IntegrateStressForElems	12.63
CalcKinematicsForElems	7.28

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

   From our analysis shown below, gprof affected the execution time. However in this case, the execution time increased insignificantly. gprof increased Real time in case of both GNU and Intel compilers by 1 second and 3 seconds, respectively. Execution time could be greater affected by gprof if more gprof options are implemented.

Compiler Name	With/Without gprof	Real time	User time	System time
GNU	Without	43.116	39.598	3.412
GNU	With	44.103	40.255	3.740
INTEL	Without	43.247	39.058	4.076
INTEL	With	46.688	41.183	5.380

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 parallel 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 the Intel compiler and the GNU compiler?**
   As shown in the tables above, there was no significant difference in the execution time between the Intel compiler and the GNU compiler, but we find differences in the execution time of specific subroutines by comparing the flat profile of GNU and Intel compilers.

4.2 Compiler Flags

4.2.1 Required Submission Files

1. **The script that automates the evaluation**
   The following files can be found in the submission folder which automates the evaluation:

   - **create_flag_comb_script.py** - A python script to automatically generate separate shell script files for gnu and intel compiler. These shell script files contain make command (**serial_runs_gcc_make.sh** and **serial_runs_icc_make.sh**) for different possible combination of gnu and intel compiler flags as mentioned in the question.
   - **extracttime.sh** - A shell script file to fetch and store only execution time values from the output history of running make for various combination of compiler flags.

Note that for this section on compiler optimisation flags, elapsed time (output from LULESH benchmark) is considered as a performance metric and speedup is defined as elapsed time (no flag)/elapsed time (with flags)

The following is performance metric and speedup sample for gnu and intel compiler optimisation flags:

(a) GNU Performance Metric & Speedup

Combination Flags	**Exec Time (s)**	**Baseline Exec Time (s)**	**Speed Up**
-march=native -fomit-frame-pointer	42.98	43.45	1.010935319
-march=native -floop-block	44.35	43.45	0.979706877
-march=native -floop-interchange	44.35	43.45	0.979706877
-march=native -floop-strip-mine	48.19	43.45	0.901639344
-march=native -funroll-loops	42.59	43.45	1.020192533
-march=native -flto	42.84	43.45	1.014239029

(b) Intel Performance Metric & Speedup

Flag Combinations	**Exec Time (s)**	**Baseline Exec Time (s)**	**Speedup**
-march=native	40.31	41.98	1.041428926
-xHost	41.52	41.98	1.011078998
-unroll	41.03	41.98	1.02315379
-ipo	43.91	41.98	0.956046459
-march=native -xHost	41.85	41.98	1.003106332

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

The Makefiles that contain the best combination of parameters for the Intel and GNU compilers are:

- GNU Compiler - Makefile_GCC (Best flag combination is -march=native -fomit-frame-pointer -funroll-loops)
- INTEL Compiler - Makefile_Intel (Best flag combination is -march=native -xHost -unroll)

4.2.2 Questions

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

icc provides approximately 117 compiler optimisation flags and gcc approximately 230

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?**

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.

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

The compiler and optimisation flags that produced the fasters binary are:

- GNU Compiler - **-march=native -fomit-frame-pointer -funroll-loops**
- INTEL Compiler - **-march=native -xHost -unroll**

4.3 Optimization Pragmas

#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

#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.

4.3.1 Required Submission Files

The following files were modified by adding the #pragma annotation

- lulesh.cc - Added the #pragma ivdep in the function CalcHourglassControlForElems

4.3.2 Questions

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

   #pragma ivdep instructs the compiler to ignore assumed vector dependencies, meaning that if the compiler sees a possible 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.

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 GNU and Intel 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;
   ```

   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. 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.

   Here is our implementation of #pragma ivdep.

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

4.4 Inline Assembler

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

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

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

4.4.2 Questions

1. **Is the 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.

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?**

   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.

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

   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.

   AVX-512 can only be used on Intel Xeon Scalable Processors or Intel Xeon Phi Processor Product Family.

   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. Other compilers supporting AVX-512 are GCC 4.9 and newer, Clang 3.9 and newer, ICC 15.0.1 and newer, Microsoft Visual Studio 2017 C++ Compiler, Java 9.

5 Performance Scaling

5.1 OpenMP

5.1.1 Required Submission Files

Refer Task 5.1 from code scripts.

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

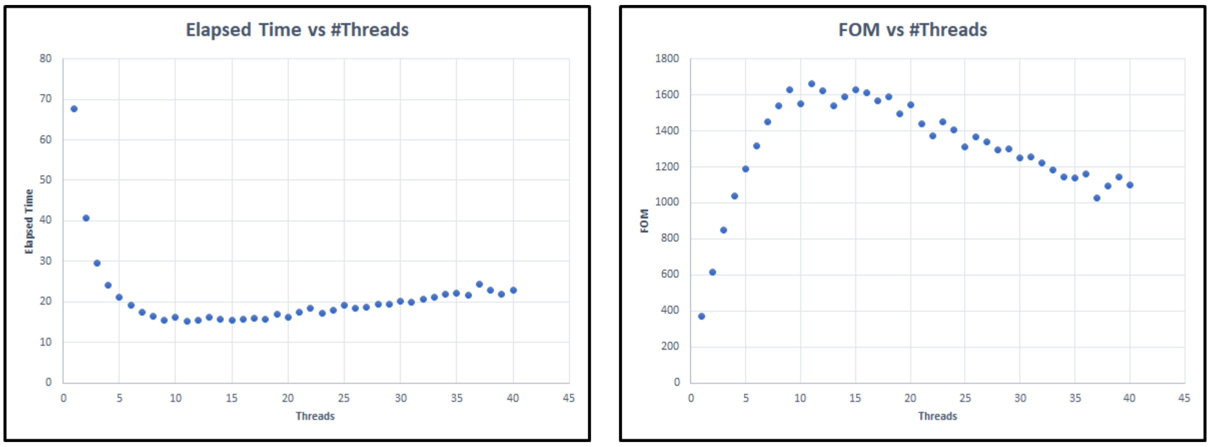


Figure 1: OpenMP in GNU compiler

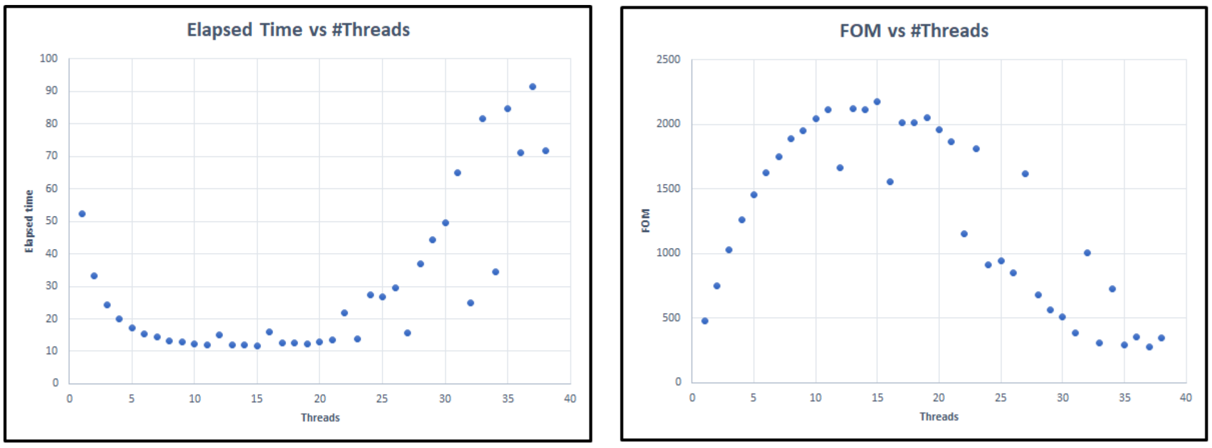


Figure 2: OpenMP in Intel compiler

2. **Copy the following information from your job script for each test:**
   Test 1:
   node =
   total_tasks =
   OMP threads =

   node = 1
   total_tasks = 1
   OMP threads = 1, 2, ..., 40

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

   In case of GNU compiler, we can notice strong scaling as FOM linearily increases upto certain number of threads, then decreases.

5.1.2 Questions

1. **Was linear scalability achieved?**

   Linear scalability was achieved for the first 11 threads in case of GNU compiler and for first 15 threads in Intel compiler after which the parallel overhead dominated and reduced the speed of the program. Once more threads were used the scalability stopped being even approximately linear and the parallel overhead began to become significant.

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

   The maximum performance was achieved at 11 threads for GNU and at 15 threads for Intel compiler. It was not the same for both compilers.

5.2 MPI

5.2.1 Required Submission Files

Refer Task 5.2 from code scripts.

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

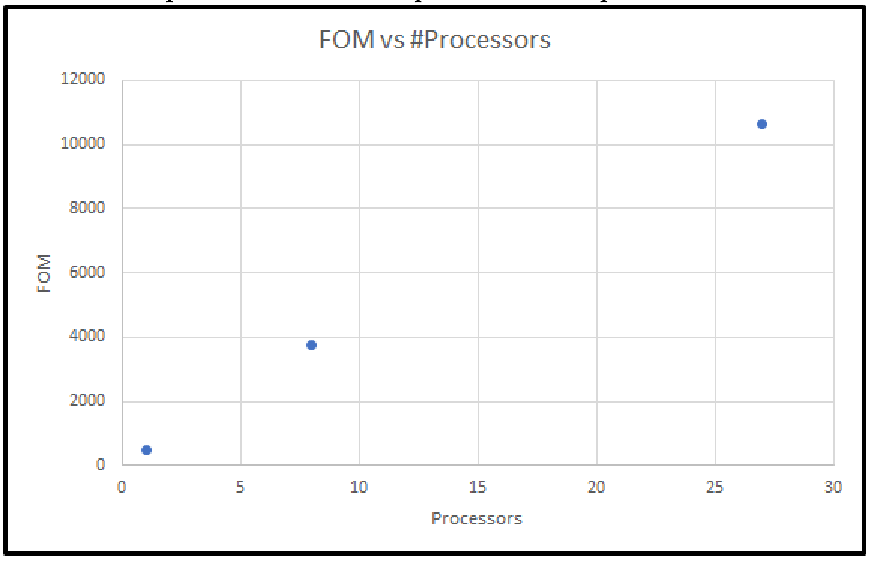


Figure 3: MPI in Intel compiler

2. **Copy the following information from your job script for each test:**
Test 1:
node =
total_tasks =

node = 1
total_tasks = 1, 8, 27

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

There is weak scaling for Intel compiler as the FOM increases with increase in number of processors.

5.2.2 Questions

1. **What are the valid combinations of processes allowed?**

   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 40 cores per node. This means that the benchmark can only be run with 1, 8 or 27 processes.

2. **Was linear scalability achieved?**

   Linear scalability was achieved.

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

   The maximum process count was achieved with 27 processors.

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

   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.

5.3 MPI + OpenMP

5.3.1 Required Submission Files

Refer Task 5.3 from code scripts.

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

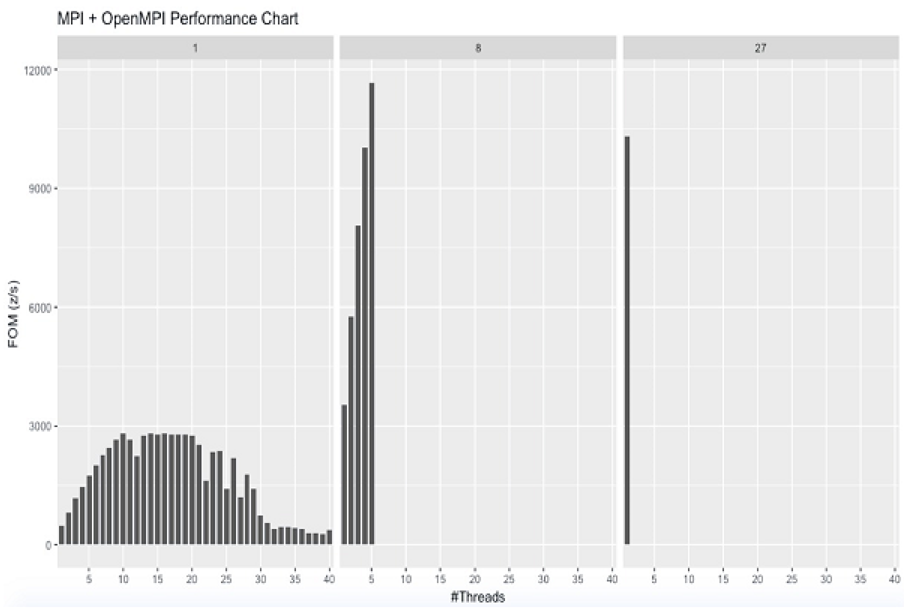


Figure 4: Hybrid in GNU compiler

2. **Copy the following information from your job script for each test:**
   Test 1:
   node =
   total_tasks =
   OMP threads =

   node = 1
   total_tasks = 1, 8, 27
   OMP threads = 1, 2, ..., 40

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

   Weak scaling is achieved as number of processors increases for Intel compiler.

5.3.2 Questions

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

   The product of the number of threads with the number of processes must be less than the number of cores on the node.

Number of processes	Possible number of threads
1	1-40
8	1-5
27	1

2. **Was linear scalability achieved?**

   Linear scalability was achieved as the number of processors increased and wasn't the case when executed on single processor.

3. **On which combination of processes and threads was the maximum performance achieved?**

   The maximum performance was seen using 8 processes and 5 threads as the best performance was achieved with the maximum number of processes, and the maximum number of threads available.

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

   These results are similar to those measured in sections 5.1 and 5.2 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. If we run the hybrid process with one thread and one process, it is slower than the OpenMP or the MPI case because there is more overhead. However, for greater numbers of processes, the hybrid helps because it combines the advantages of both MPI and OpenMP.

5. **Which solution is overall the fastest?**

   The fastest solution on the phase 1 node was OpenMP/MPI with 8 processes and 5 threads. 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.

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

   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 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.

6 Contribution

1. **Smith**
   Actively worked on coding section of the assignment and report.
   Created an automated script for evaluating the benchmark with different compiler flag combinations.
   Building the benchmark in multi threaded mode (MPI + OpenMP).

2. **Shyam**
   Understanding of the assignment, setup and Final report creation.
   Worked on building the benchmark in single threaded mode.
   Building the benchmark in multi threaded mode (MPI).

3. **Siddhesh**
   Building the benchmark in multi threaded mode (Open MP).
   Understanding GNU profiler, Inline assemblers and their usage.

4. **Ulugbek**
   Understanding the performance effect of available optimization flags with GNU and Intel compilers.
   Understanding Optimization pragmas and their application.