Assignment1 - Single Node Performance

John Kouroudis, Patrick O'Connor and William Parker

November 2017

1 Single-thread Performance

1.1 GPROF

GPROF was used to analyse potential parts of the code that could be optimised and to find bottlenecks. Two GPROF profiles were generated one for the Phase one architecture on the thin nodes using Sandy Bridge and another for the Phase two architure on the Haswell platform. It was found by analysing the generated flat profile for the file assignment1.phase1.gprof.out that there were no routines which took 80% or more of the execution time. Similarly for phase two by analysing the flat profile found in assignment1.phase2.gprof.out there were no routines that required 80% or more of the execution time. Results of significant time spent in a routine can be found in Table 1.

It was expected that the program would run slower with the gprof compiler flags enabled due to the time spent collecting and writing profile data. It was observed using the time command that the program ran 1 second slower with the gprof flags enabled Table 2.

Gprof is not capable of analysing various loops, it is function level profiling and uses statistical sampling in order to calculate its timing values. The overhead caused by Gprof is quite high from 30-260% therefore it is concluded that it is not suitable for long running applications. Gprof is not capable of analysing parallel applications. In order to profile parallel applications a profiling tool such as Valgrind, LIKWID, or gperftools should be used. It was observed that between the two different nodes, Phase one and Phase two that the time spent in functions varied although the functions the time was spent in was the same Table 1. From Table 1 it can be seen that the phase 2 nodes were able to run the benchmark $\approx$ 8 seconds quicker than phase 1.

1.2 Compiler Flags

We obtained errors when attempting to compile using gcc 4.9 and all of the compiler flags, so we used gcc 5.0 instead which did not produce any errors. The grind time was identified as the performance metric of the benchmark, providing the number of updates per element per second. Two python scripts were written in order to automate this part of the task. One to compile and submit jobs to the superMUC (compiler_flags.py) and another to parse the results (parse_results.py). The Makefile with the best compiler flag combination for the Phase 1 nodes is found in results/task4.2.1/intel/sb/results/Makefile this ended up just being the **-unroll** flag. The corresponding Makefile

Routine Name	Percentage time spent	time spent / s	Phase
ApplyMaterialPropertiesForElems	54.78	12.78	1
CalcElemShapeFunctionDerivatives	26.53	6.19	1
CalcKinematicsForElems	14.02	3.27	1
ApplyMaterialPropertiesForElems	51.58	9.80	2
CalcElemShapeFunctionDerivatives	26.89	5.11	2
CalcKinematicsForElems	17.42	3.31	2

Table 1: Runtimes of several routines as determined by gprof on varying architectures of SuperMUC

Time / s	GPROF compiler flags enabled	Phase
0m39.420s	no	1
0m40.057s	yes	1
0m32.864s	no	2
0m33.744 s	yes	2

Table 2: Runtimes of the binary with and without gprof flags.

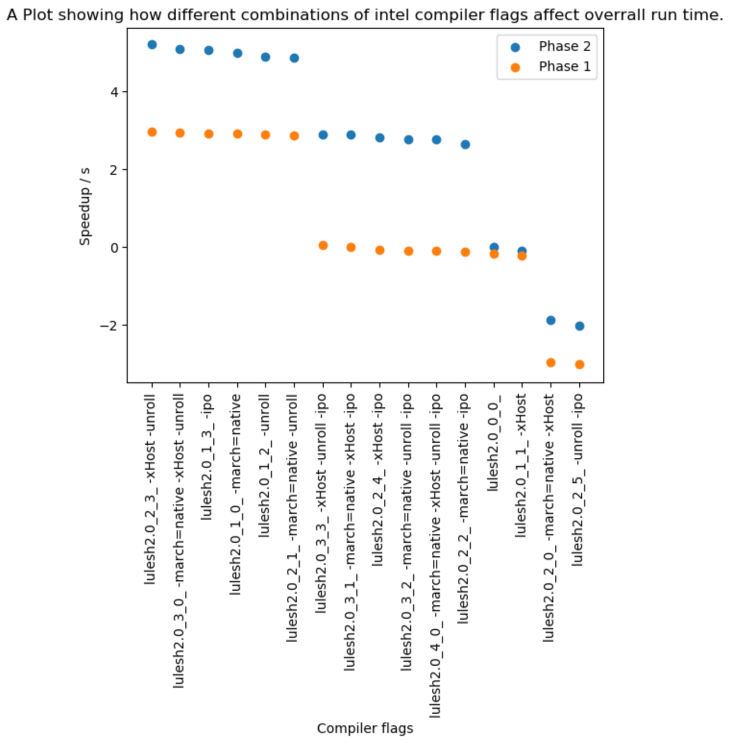

Figure 1: Compiler flags and corresponding speedups for the Intel compiler.

for the best phase 2 node is found in results/task4.2.1/lulesh2.0/Makefile and consisted of **-xHost** and **-unroll**. The full results of the speedups for both compilers can be seen from Figures 1 & 2.

For the gcc compiler there were approximately 229 optimisation flags and for icc compiler there are approximately 20 (https://software.intel.com/en-us/node/522802). Given the time it took to evaluate all of the different compiler flag options, it is not realistic to evaluate all the different combinations. This will depend on the problem being solved in our case we can approximate

$$\frac{128\text{combinations} \times 40\text{avg. seconds / combination}}{60\text{seconds}} \approx 85\text{minutes}.$$

It was found that the quickest compiler was the Intel compiler on the Phase 2 architecture with the following flags **-xHost -unroll** with an overall runtime of 28.77 seconds giving a speedup of 5.22 seconds compared to the measured baseline. The corresponding Makefile is found in results/task4.2.1/lulesh2.0/Makefile.

1.3 Optimisation Pragmas

Pragmas are directives that allow the programmer to provide instructions or hints to the compiler for use in specific cases, or parts of the code. Firstly, the following GCC pragmas and their functions were investigated:

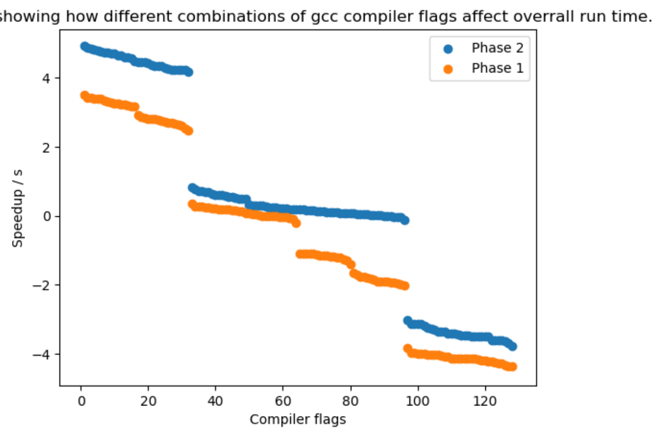

Figure 2: Compiler flags and corresponding speedups for the gcc compiler. The numbers on the x axis correspond to a combination of compiler flags as shown in Table 4.

- #pragma GCC ivdep - The programmer asserts that there are no loop-carried dependencies which would prevent consecutive iterations of the following loop from executing concurrently with SIMD (single instruction multiple data) instructions.
- #pragma optimize(”string”...) - allows you to set global optimisation options for every function that follows this declaration. Optimisation options include inline, noinline, aligned and other.

Secondly, the following Intel pragmas and their functions were investigated:

- #pragma simd - Used to guide the compiler to vectorise more loops. Vectorisation using the simd pragma complements (but does not replace) the fully automatic approach.
- #pragma ivdep instructs the compiler to ignore assumed vector dependencies. To ensure correct code, the compiler treats an assumed dependence as a proven dependence, which prevents vectorisation. This pragma overrides that decision. The loop count pragma specifies the minimum, maximum, or average number of iterations for a for loop. In addition, a list of commonly occurring values can be specified to help the compiler generate multiple versions and perform complete unrolling. The vector pragma indicates that the loop should be vectorised, if it is legal to do so, ignoring normal heuristic decisions about profitability.
- #pragma inline - specifies that a subprogram call is, or is not to be inlined.
- #pragma noinline - Declares a function whose inline expansion is to be stopped when the noinline level option is used.
- #pragma forceinline - Indicates that the calls in question should be inlined whenever the compiler is capable of doing so.
- #pragma nounroll / unroll - Tells the compiler that the register pressure is either sufficiently small or large that the loop unrolling optimisation might or might not be profitable.
- #pragma unroll_and_jam - Partially unrolls one or more loops higher in the nest than the innermost loop and fuses/jams the resulting loops back together. This transformation allows more reuses in the loop.
- #pragma nofusion - Allows you to fine tune your program on a loop-by-loop basis. This pragma should be placed immediately before the loop that should not be fused.
- #pragma distribute_point - Used to suggest to the compiler to split large loops into smaller ones; this is particularly useful in cases where optimisations like vectorisation cannot take place due to excessive register usage.

In answer to the second part of question 4.3.1 the difference of those pragmas is the severity with which the optimisation is enforced. pragma simd always enforces vectorisation of loop, regardless of cost or safety. pragma vector prompts the compiler to ignore efficiency heuristics when deciding on vectorisation. Code under this command might be slower than without it, but always safe. pragma ivdep tells compiler to ignore assumed data dependencies that inhibit vectorisation (for example loop carried dependencies), but not proven ones. A common example are pointers, as under this directive the compiler will assume they are not pointing to the same memory location and vectorise. However, clear dependencies will not be ignored with this directive as they would with pragma simd. For this part of the report, it was required identify a function which consumed the most substantial amount of computational time and apply the best pragma directive to optimise it. Judging from our grpof run we first looked at Apply element materials but were not able to gain any notable speedups. After conferring with some other groups we decided to look at CalcHourglassControlForElems. Our grpof results never returned this function as expensive, despite repeated attempts (not run on the login node) but many others did so we carried on with our optimisation efforts here.

The function CalcHourglassControlForElems consists of a loop over all the elements of the domain, which are a considerable amount. Within this loop, a multitude of diverse calculations take place and therefore any sort of vectorisation or pipelining optimisation is difficult. To counter this, the gcc pragma ivdep was used. This instructed the compiler that there are no dependencies and therefore the loop can be safely broken up and vectorised. As is evident by the experiment, this noticeably decreases the lulesh performance metric, increasing the performance. Using the Intel compiler, the pragma distribute point was used. This directive is used to suggest to the compiler to split up large loops into smaller ones, thus relieving pressure on the registers and allowing further optimisations to take place.

1.4 Inline Assembler

An inline assembler is a feature held by some compilers that permit the injection of low level assembly code into a program that has been written and compiled from a higher level language like c or c++. This would be done normally in one of three situations: Optimisation, accessing processor instructions and the ability to make system calls. An example of Intel inline assembler code is

```
at&t noprefix                      intel
mov eax, -4(ebp,edx,4)             mov DWORD PTR[-4 +ebp +edx *4], eax
mov eax, -4(ebp)                   mov DWORD PTR[-4 +ebp], eax
mov edx, (ecx)                     mov DWORD PTR[ecx], edx
lea (    ,eax,4), eax              lea eax, DWORD PTR[8 + eax*4]
lea (eax,eax,2), eax               lea eax, DWORD PTR[eax*2+eax]
```

Inline assembler is not necessarily faster than compiler generated code. It is entirely dependent on the code / algorithm being written. It is possible to produce an example where inline assembler code will be better than compiled code but most of the time the compiler is able to do a more than adequate job of producing good assembly code. As indicated above it is possible to harness new CPU instructions using inline assembler if the compiler is not able to handle them yet.

AVX-512 are 512-bit extensions to the 256-bit Advanced Vector Extensions SIMD instructions for x86 instruction set architecture. It is currently supported by Intel's Xeon Phi x200 and Skylake-X core i7 and i9 models.

2 Multi-thread Performance

2.1 OpenMP

We determined that the most important metric was the overall grind time. Figure 3 shows results on both phases and that linear scalabilty was achieved as the threads were increased from 1-8, after 8 the increased in performance of the benchmark as measured by the overall grind time was not measured to be linear. The maximum speedup for Haswell with n=28 threads/core was with n=16, runtime=8.6s, speedup=8.60, and efficiency=0.23. Sandy Bridge with n=16 threads/core attained maximum speedup at n=13, runtime=8.33s, speedup=4.12 and efficiency=0.32.

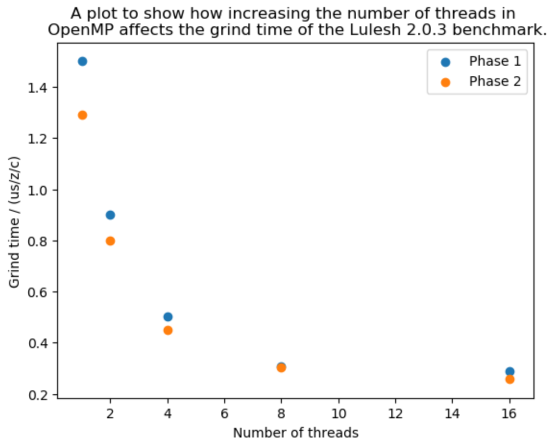

Figure 3: A plot showing the decrease in overall grind time as the number of threads used by OpenMP is increased from 1-8 threads. Linear scalability held up to this point and after 8 threads levelled out.

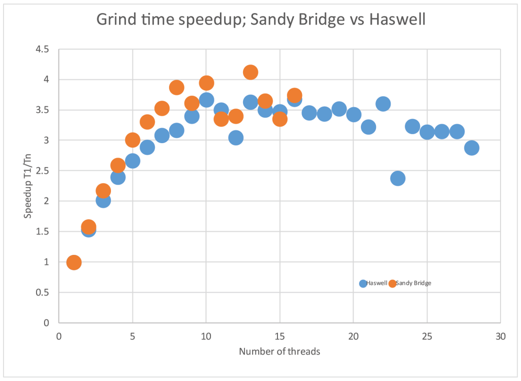

Figure 4: A computation of speedup using the grind time.

2.1.1 Amdahl's Law investigation

OpenMP is a cross platform set of programming tools which use global shared memory address space to parallelise a kernel among threads. OpenMP uses high level directive base pragmas to implement a fork join concept of parallelism. Underneath the hood though, the high level OpenMP pragmas rely on posix pthread calls. The Lulesh kernel was compiled with an intel 16.0 compiler using optimised flags with OpenMP on both sandy bridge and haswell architectures. The executable was run using a range of threads from 1 to either 16 or 28 threads depending on the architecture. Figure 4 shows the grind time speedup on a single core vs the number of threads for different architectures. The results were consistent with Amdahl's law. Some differences in architecture performance were observed.

2.2 MPI

Another form of parallelism is called MPI (message passing interface). This, in contrast to the OpenMP model where threads have direct access to shared memory address space. With MPI there is no communal memory and each time a value from a domain is required from a nearby processor, it is explicitly sent and received. This process along with the initial setup of the domains is the only significant overhead of MPI. To minimise the amount of data needed to be communicated the domain decomposition cannot be arbitrary. To that effect, lulesh always divides the domain in cubes, thus ensuring maximum coverage with minimum surface and thus communication. This constrains the number of processors to a cube of an integer.

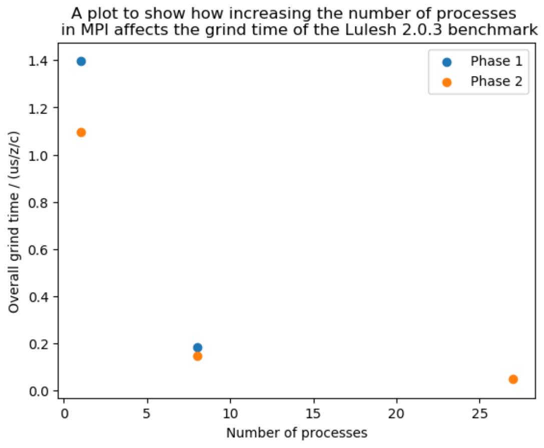

Figure 5: A graph showing how varying the number of processors affects the overall grind time as measured by the lulesh benchmark.

To satisfy this condition and taking into account hardware constrains (a maximum of 16 processors for phase 1 and 28 for phase 2), experiments were run with 1 and 8 cores for phase 1 and 1,8 and 27 for phase 2. This led to the following results. It must be noted, that lulesh generates one domain of size 27000 for every processor. This means that time is no longer an equivalent metric to Grind Time, and the overall Grind Time is needed.

Figure 5 shows the how the number of processors relates to the overall grind time. Although it is difficult to extrapolate linearity from that amount of data, the curves seem indeed to suggest that, eventually reaching a plateau which is in agreement with Amdahl's law.

The best number of MPI tasks for phase 1 is 8 and phase 2 is 27. Although absolute performance increased, efficiency drops, which needs to be taken into account for effective resource management. Therefore, assuming a plateau is reached, the number of processors that best balances between speedup and efficiency is 8 for both phases. Conclusively, although more MPI tasks consistently produce better grind times for the lulesh benchmark, the balance between speedup and efficiency dictates the choice of number of MPI tasks.

MPI performs consistently better than openmp. This is in agreement with lulesh documentation stating that the communication overhead is at most 10%, even for large systems. In contrast to this, openmp requires a run-time overhead due to scheduling, private variable creation and dormant processors. Furthermore, openmp relies on memory that is not local to the processors and each thread acquires data by direct communication with that memory. MPI on the other hand sets up an independent domain in the local cache of the processors, only requiring communication on the boundaries. In this case, the overhead to access the shared memory has proven to be more significant than the MPI pitfalls, although of course, both provide a substantial improvement on performance in comparison to the sequential implementation.

2.3 MPI + OpenMP

In this section, a combination of MPI and OpenMP was utilised. Effectively, the domain was partitioned in multiple sub domains, each of which was handled by the allocated MPI processor. This processor then distributes the workload among multiple threads, thus improving performance minimising the communication costs.

The constrains remain the same as in the previous sections. OpenMP allows any number of threads, while MPI processors must always be cubes of integers. Additionally, the number of threads*number of MPI tasks must be lower or equal the maximum number of processors of the given phase. Table 3 shows the following valid combinations for the different phases.

As can be seen in Figure 6 linear scalablity was approximately achieved initially, degrading into a plateau as is consistent with Amdahl's law. From the graphs, it is obvious that the best balance between performance and efficiency is found at 1 thread and 8 processes for phase 1 and phase 2. Exceeding those points, absolute performance again increases, but efficiency drops.

No. Processors	No. Threads	Phase
1	1	1 & 2
1	2	1 & 2
1	4	1 & 2
1	8	1 & 2
1	16	1 & 2
8	1	1 & 2
8	2	1 & 2
27	1	2

Table 3: A table showing the valid combinations of threads and processors for the different phases.

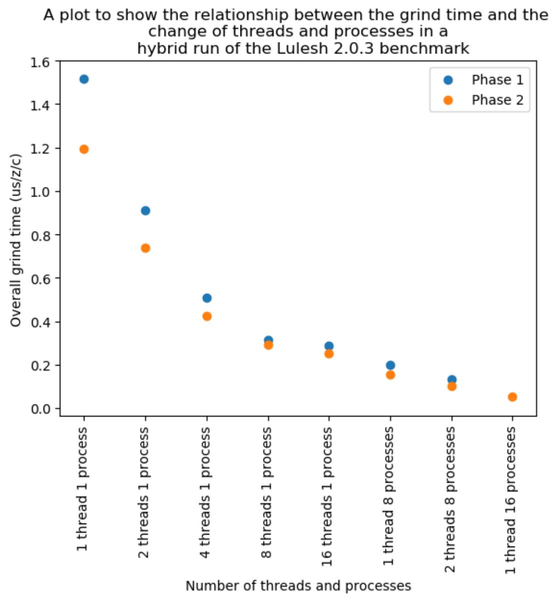

Figure 6: A graph showing overall grind time as numbers of threads and processors are varied.

This proved to be the most efficient use of hardware among openmp, MPI and hybrid. It is not entirely surprising, as lulesh has been proven to scale well with MPI, with a small communication overhead. OpenMP however needs substantial run-time scheduling and variable creation which, doesn't provide as efficient a use of those threads. This is evident by the hybrid grind time, where for equivalent number of processors, the measurement with the largest number of MPI tasks is faster. Comparing 16 threads 1 process to 8 processes 1 thread, it is evident that even with fewer resources allocated to MPI, the performance is increased.

The results are not overly surprising, as the lulesh benchmark is specifically optimised with respect to the MPI communication overhead. In contrast, openmp suffers from the constant run-time need of each thread accessing the shared memory, scheduling overhead and threads queuing for one task. Specifically for the SuperMUC architecture, this has proven to be more cumbersome than message parsing of boundary values and therefore the best use of the available hardware is using it for the MPI implementation of the lulesh benchmark.

A GCC compiler flag key

ID	Flags
1	-march=native -fomit-frame-pointer -floop-block -floop-interchange -floop-strip-mine -funroll-loops
2	-march=native -floop-interchange -floop-strip-mine -funroll-loops
3	-march=native -floop-block -floop-strip-mine -funroll-loops
4	-march=native -fomit-frame-pointer -floop-interchange -funroll-loops
5	-march=native -floop-block -floop-interchange -funroll-loops
6	-march=native -fomit-frame-pointer -floop-block -funroll-loops
7	-march=native -floop-strip-mine -funroll-loops
8	-march=native -fomit-frame-pointer -floop-interchange -floop-strip-mine -funroll-loops
9	-march=native -floop-block -floop-interchange -floop-strip-mine -funroll-loops
10	-march=native -floop-interchange -funroll-loops
11	-march=native -funroll-loops
12	-march=native -fomit-frame-pointer -funroll-loops
13	-march=native -fomit-frame-pointer -floop-block -floop-strip-mine -funroll-loops
14	-march=native -fomit-frame-pointer -floop-block -floop-interchange -funroll-loops
15	-march=native -floop-block -funroll-loops
16	-march=native -fomit-frame-pointer -floop-interchange -floop-strip-mine
17	-floop-block
18	-march=native -floop-strip-mine
19	-march=native -fomit-frame-pointer -floop-strip-mine -funroll-loops
20	-march=native -fomit-frame-pointer -floop-block -floop-interchange
21	-march=native -floop-interchange -floop-strip-mine
22	-march=native -floop-block -floop-interchange
23	-march=native
24	-march=native -fomit-frame-pointer -floop-block -floop-interchange -floop-strip-mine
25	-march=native -fomit-frame-pointer -floop-block -floop-strip-mine
26	-march=native -floop-block -floop-interchange -floop-strip-mine
27	-march=native -floop-block
28	-march=native -floop-interchange
29	-march=native -fomit-frame-pointer -floop-strip-mine
30	-march=native -fomit-frame-pointer -floop-interchange
31	-march=native -fomit-frame-pointer -floop-block
32	-march=native -floop-block -floop-strip-mine
33	-march=native -funroll-loops -flto
34	-march=native -floop-block -floop-interchange -floop-strip-mine -funroll-loops -flto
35	-march=native -floop-block -floop-interchange -funroll-loops -flto
36	-march=native -fomit-frame-pointer -floop-block -floop-interchange -funroll-loops -flto
37	-march=native -floop-interchange -funroll-loops -flto
38	-march=native -floop-strip-mine -funroll-loops -flto
39	-march=native -fomit-frame-pointer -funroll-loops -flto
40	-march=native -fomit-frame-pointer -floop-strip-mine -funroll-loops -flto
41	-march=native -floop-interchange -floop-strip-mine -funroll-loops -flto
42	-march=native -fomit-frame-pointer -floop-block -floop-interchange -floop-strip-mine -funroll-loops -flto
43	-march=native -floop-block -funroll-loops -flto
44	-march=native -fomit-frame-pointer -floop-interchange -funroll-loops -flto
45	-march=native -fomit-frame-pointer -floop-block -floop-strip-mine -funroll-loops -flto

46	-march=native -floop-block -floop-strip-mine -funroll-loops -flto
47	-march=native -fomit-frame-pointer -floop-block -funroll-loops -flto
48	-march=native -fomit-frame-pointer -floop-interchange -floop-strip-mine -funroll-loops -flto
49	-floop-block -floop-interchange -floop-strip-mine -funroll-loops
50	-march=native -floop-interchange -flto
51	-march=native -fomit-frame-pointer -floop-block -floop-interchange -floop-strip-mine -flto
52	-march=native -fomit-frame-pointer -floop-block -flto
53	-march=native -floop-block -floop-interchange -floop-strip-mine -flto
54	-fomit-frame-pointer -floop-interchange -floop-strip-mine
55	-march=native -floop-block -floop-strip-mine -flto
56	-march=native -fomit-frame-pointer -floop-block -floop-strip-mine -flto
57	-march=native -fomit-frame-pointer -floop-interchange -floop-strip-mine -flto
58	-march=native -floop-interchange -floop-strip-mine -flto
59	-floop-block -floop-interchange -floop-strip-mine
60	-march=native -floop-block -flto
61	-fomit-frame-pointer -floop-block -floop-strip-mine
62	-march=native -floop-strip-mine -flto
63	-march=native -fomit-frame-pointer -floop-strip-mine -flto
64	-march=native -flto
65	-march=native -fomit-frame-pointer -floop-block -floop-interchange -flto
66	-fomit-frame-pointer -floop-block -floop-interchange -floop-strip-mine
67	-fomit-frame-pointer -floop-block -floop-interchange -funroll-loops
68	-fomit-frame-pointer -floop-block -floop-interchange
69	-fomit-frame-pointer -floop-block -floop-strip-mine -funroll-loops
70	-floop-block -floop-strip-mine -funroll-loops
71	-march=native -fomit-frame-pointer -floop-interchange -flto
72	-fomit-frame-pointer -floop-block -floop-interchange -floop-strip-mine -funroll-loops
73	-floop-strip-mine
74	-fomit-frame-pointer -floop-block
75	-march=native -floop-block -floop-interchange -flto
76	-flto
77	-fomit-frame-pointer -floop-interchange -funroll-loops
78	-floop-interchange
79	-fomit-frame-pointer -floop-strip-mine
80	-funroll-loops
81	-fomit-frame-pointer
82	-floop-interchange -floop-strip-mine -funroll-loops
83	-fomit-frame-pointer -floop-interchange -floop-strip-mine -funroll-loops
84	-floop-block -floop-strip-mine
85	-march=native -fomit-frame-pointer -flto
86	-floop-block -funroll-loops
87	-fomit-frame-pointer -floop-interchange
88	-floop-interchange -floop-strip-mine
89	-floop-block -floop-interchange
90	
91	-fomit-frame-pointer -floop-strip-mine -funroll-loops
92	-floop-block -floop-interchange -funroll-loops
93	-fomit-frame-pointer -floop-block -funroll-loops
94	-floop-strip-mine -funroll-loops
95	-fomit-frame-pointer -funroll-loops
96	-floop-interchange -funroll-loops
97	-fomit-frame-pointer -floop-interchange -floop-strip-mine -flto
98	-fomit-frame-pointer -floop-block -floop-interchange -flto
99	-fomit-frame-pointer -floop-interchange -flto

100	-fomit-frame-pointer -floop-block -floop-interchange -floop-strip-mine -flto
101	-floop-block -floop-interchange -flto
102	-fomit-frame-pointer -floop-interchange -funroll-loops -flto
103	-floop-block -floop-strip-mine -flto
104	-fomit-frame-pointer -floop-block -floop-strip-mine -flto
105	-floop-block -flto
106	-fomit-frame-pointer -floop-block -flto
107	-fomit-frame-pointer -floop-block -floop-interchange -floop-strip-mine -funroll-loops -flto
108	-floop-strip-mine -funroll-loops -flto
109	-floop-block -funroll-loops -flto
110	-fomit-frame-pointer -floop-strip-mine -flto
111	-floop-strip-mine -flto
112	-floop-interchange -flto
113	-fomit-frame-pointer -floop-interchange -floop-strip-mine -funroll-loops -flto
114	-fomit-frame-pointer -flto
115	-march=native -fomit-frame-pointer
116	-fomit-frame-pointer -floop-block -floop-interchange -funroll-loops -flto
117	-fomit-frame-pointer -floop-block -floop-strip-mine -funroll-loops -flto
118	-floop-interchange -floop-strip-mine -flto
119	-floop-block -floop-interchange -floop-strip-mine -flto
120	-floop-block -floop-interchange -floop-strip-mine -funroll-loops -flto
121	-fomit-frame-pointer -floop-strip-mine -funroll-loops -flto
122	-fomit-frame-pointer -funroll-loops -flto
123	-floop-block -floop-interchange -funroll-loops -flto
124	-floop-block -floop-strip-mine -funroll-loops -flto
125	-floop-interchange -funroll-loops -flto
126	-floop-interchange -floop-strip-mine -funroll-loops -flto
127	-fomit-frame-pointer -floop-block -funroll-loops -flto
128	-funroll-loops -flto

Table 4: A table showing the valid combinations of threads and processors for the different phases.