Efficient Implementation of 3D Finite Difference Schemes on Recent Processor Architectures

FREDERICK CEDER
Efficient Implementation of 3D Finite Difference Schemes on Recent Processor Architectures

Effektiv implementering av finita differensmetoder i 3D på senaste processorarkitekturer

Frederick Ceder
fceder@kth.se

Master Thesis in Computer Science 30 Credits
Examensarbete i Datalogi 30 HP

Supervisor: Michael Schliephake
Examinator: Erwin Laure

Computer Science Department
Royal Institute of Technology
Sweden

June 26, 2015
ABSTRACT

In this paper a solver is introduced that solves a problem set modelled by the Burgers equation using the finite difference method: forward in time and central in space. The solver is parallelized and optimized for Intel Xeon Phi 7120P as well as Intel Xeon E5-2699v3 processors to investigate differences in terms of performance between the two architectures.

Optimized data access and layout have been implemented to ensure good cache utilization. Loop tiling strategies are used to adjust data access with respect to the L2 cache size. Compiler hints describing aligned memory access are used to support vectorization on both processors. Additionally, prefetching strategies and streaming stores have been evaluated for the Intel Xeon Phi. Parallelization was done using OpenMP and MPI.

The parallelisation for native execution on Xeon Phi is based on OpenMP and yielded a raw performance of nearly 100 GFLOP/s, reaching a speedup of almost 50 at a 83% parallel efficiency. An OpenMP implementation on the E5-2699v3 (Haswell) processors produced up to 292 GFLOP/s, reaching a speedup of almost 31 at a 85% parallel efficiency. For comparison a mixed implementation using interleaved communications with computations reached 267 GFLOP/s at a speedup of 28 with a 87% parallel efficiency. Running a pure MPI implementation on the PDC’s Beskow supercomputer with 16 nodes yielded a total performance of 1450 GFLOP/s and for a larger problem set it yielded a total of 2325 GFLOP/s, reaching a speedup and parallel efficiency at resp. 170 and 33.3% and 290 and 56%.

An analysis based on the roofline performance model shows that the computations were memory bound to the L2 cache bandwidth, suggesting good L2 cache utilization for both the Haswell and the Xeon Phi’s architectures. Xeon Phi performance can probably be improved by also using MPI. Keeping technological progress for computational cores in the Haswell processor in mind for the comparison, both processors perform well. Improving the stencil computations to a more compiler friendly form might improve performance more, as the compiler can possibly optimize more for the target platform. The experiments on the Cray system Beskow showed an increased
efficiency from 33.3% to 56% for the larger problem, illustrating good weak scaling. This suggests that problem sizes should increase accordingly for larger number of nodes in order to achieve high efficiency.
SAMMANFATTNING


Vi optimerade programmet med omtanke på dataåtkomst och minneslayout för att få bra cacheutnyttjande. Loopblockningsstrategier används också för att dela upp arbetsminnet i mindre delar för att begränsa delarna i L2 cacheminnet. För att utnyttja vektorisering till fullo så används kompilatordirektiv som beskriver minnesåtkomsten, vilket ska hjälpa kompilatorn att förstå vilka dataaccesser som är alignade. Vi implementerade också prefetching strategier och streaming stores på Xeon Phi och disskuterar deras värde. Paralleliseringen gjordes med OpenMP och MPI.

Paralleliseringen för Xeon Phi:en är baserad på bara OpenMP och exekverades direkt på chipet. Detta gav en rå prestanda på nästan 100 GFLOP/s och nådde en speedup på 50 med en 83% effektivitet. En OpenMP implementation på E5-2699v3 (Haswell) processorn fick upp till 292 GFLOP/s och nådde en speedup på 31 med en effektivitet på 85%. I jämförelse fick en hybrid implementation 267 GFLOP/s och nådde en speedup på 28 med en effektivitet på 87%. En ren MPI implementation på PDC’s Beskow superdator med 16 noder gav en total prestanda på 1450 GFLOP/s och för en större problemställning gav det totalt 2325 GFLOP/s, med speedup och effektivitet på respektive 170 och 33% och 290 och 56%.

En analys baserad på roofline modellen visade att beräkningarna var minnesbudna till L2 cache bandbredden, vilket tyder på bra L2-cache användning för både Haswell och Xeon Phi:s arkitekturer. Xeon Phis prestanda kan förmodligen förbättras genom att även använda MPI. Håller man i åtanke de tekniska framstegen när det gäller beräkningskärnor på de senaste åren, så presterar både arkitekturer bra. Beräkningskärnan av implementationen kan förmodligen anpassas till en mer kompila-
torvänlig variant, vilket eventuellt kan leda till mer optimeringar av kompilatorn för respektive plattform. Experimenten på Cray-systemet Beskow visade en ökad effektivitet från 33,3% till 56% för större problemställningar, vilket visar tecken på bra weak scaling. Detta tyder på att effektivitet kan uppehållas om problemställningen växer med fler antal beräkningsnoder.
CONTENTS

1 INTRODUCTION 1
   1.1 Purpose 1
   1.2 Scope of Thesis 1
   1.3 Value of project 2

2 MODERN PROCESSOR ARCHITECTURE 3
   2.1 Introducing Xeon E5-2699 v3 and Xeon Phi 3
   2.2 In-Order and Out-of-Order Core 4
   2.3 Vectorization 7
   2.4 Streaming Stores 16
   2.5 Cache Hierarchy 18
   2.6 Prefetching 19

3 MATHEMATICAL BACKGROUND 23
   3.1 Burgers Equation Background 23
   3.2 The Fictional Flow Problem 24
   3.3 Finite Difference Method 24
   3.4 Explicit and Implicit FDMs 28
   3.5 Explicit Scheme: FTCS scheme 29
   3.6 Stability: Explicit vs Implicit 34
   3.7 Accuracy: Explicit vs Implicit 34

4 RELATED WORK 37
5 IMPLEMENTATION 39
   5.1 Program execution 40
   5.2 Initialization Phase 44
   5.3 Update Algorithm 44
   5.4 Sequential Version 48
   5.5 Parallel Version 61

6 BENCHMARK 83
   6.1 Test platforms 83
   6.2 Problem Set 84
   6.3 Compilation 84
   6.4 Runtime Configurations 86
   6.5 Timing 86
   6.6 Thread and Process Control 87
   6.7 Reference Times 92
   6.8 Speedup and Efficiency Graphs 93
   6.9 Roofline Model 93

7 RESULTS 97
   7.1 Roofline Models 97
INTRODUCTION

In this paper a strategy for implementing a 3D finite difference scheme solver on recent processor architectures is presented. The Burgers equation is implemented and parallelized on a more conventional CPU cluster and on Intel’s newest MIC architecture. The following sections describes the purpose, scope and value of the project.

1.1 PURPOSE

The purpose of this thesis is to find strategies on how to make a 3D difference scheme solver of higher order to run efficiently on modern processor architectures and how to parallelize it. Intel’s new Xeon Phi Knights Corner is based on a highly parallel coprocessor architecture, Many-Integrated-Core (MIC), that can run up to 244 hardware threads in parallel. It features wider vector registers than other Intel processors, which increases a single cores computational capabilities of floating-point values. This paper utilizes these new features by using portable compiler hints or pragmas to utilize core architectural features and widely used APIs such as OpenMP and MPI, without explicitly coding platform specific features like for example intrinsics. In practice this means that the optimization and parallelization strategies applied here can with minor tweaks be utilized on other x86 Intel architectures and gain speedup.

1.2 SCOPE OF THESIS

The scope of the thesis is to implement a higher order difference scheme solver with minor discussion about accuracy and stability of the method to be applied. The solver will be optimized in terms of data layout, data access and cache optimizations.

1 Many-Integrated-Core (MIC)
2 Message Passing Interface (MPI)
for Intel processor architectures. Appropriate Intel compiler hints will be used, allowing the compiler to auto-vectorize parts of code for the hosts vectorization unit, speeding up the scalar computations done in the implementation. For the current MIC model this implies 512-bit wide vector registers in the VPU, yet it will also work with minor tweaks for systems with other instruction sets, such as AVX or AVX-2. A mix of OpenMP and MPI will be used to parallelize code. As cluster computing is widely used to solve large scale problem, the MPI-based implementations will also be evaluated on a larger cluster environments.

1.3 VALUE OF PROJECT

The Burgers model is a simplification of the Navier-Stokes model \cite{28} and thus the chosen solver uses a similar computational kernel as a Navier-Stokes model implementation. The Navier-Stokes equations describe the motion of viscous fluids and can be used to model weather, ocean currents, water flow in a pipe or air flow around objects. Also, structured meshes are very common in many other contexts and offers the reader a mild introduction to explicit numerical methods. The contribution of this paper is a suggestion of optimization strategies that can be implemented on both Haswell and Xeon Phi processor to produce portable and efficient code. Any model using a similar structured mesh and computational kernel will be able to benefit from the suggestions and observations made in this paper.
The Intel Xeon Phi Knights Corner was released in 2012 and is the first MIC architecture chip produced and the model referred to in this writing is 'Xeon Phi Coprocessor 7120P'. The Haswell chips are a processor family that were announced around 2012 and the Xeon reference model that is going to be referred to is the 'Xeon E5-2699 v3' from 2014. Both processors extend the x86 Instruction-Set (x86) and are thus based on the same Instruction Set Architecture (ISA). Both designs share many architectural features and design choices that are described in the rest of this section. What follows is a simplified specification of the Xeon Phi compared to the Xeon E5-2699 v3, where the following information will be relevant to design choices in the implementation referred to in chapter 5.

For a more detailed specification the reader is urged to study the System Software Developer’s Guide for each respective processor or any other source listed in the bibliography section more carefully.

All compiler switches references made refer to the linux intel compiler as of version 15.0.2. The intel compiler on Windows has in most cases the same switches, however slightly different names.

2.1 INTRODUCING XEON E5-2699 V3 AND XEON PHI

The Intel Xeon Phi is a SMP coprocessor on a single card that can be connected to a PCI Express port of a host system. The host processor of a system will communicate with the Xeon Phi over the PCI-bus and can offload computations or entire applications, which are to be executed, to it. The Xeon E5-2699 v3 is a Haswell chip and is a main processor, which means that the chips does not have to be connected to a host system in order to function.

1 Symmetric Multi-Processing (SMP)
The Intel Xeon Phi Coprocessor 7120P has a MIC architecture, with 61 in-order cores clocked at 1.238GHz. Each core features a VPU, an L2 cache globally shared with the other cores over the ring interconnect and has four hardware thread execution contexts. The ring interconnect connects all the coprocessors components and provides a flexible communication channel. In total there are eight on-die GDDR5-based memory controllers, each having two memory channels that have in total an aggregated theoretical bandwidth of 352 GB/s or 44 GB/s per controller to the 16 GB RAM. [10, Sec. 2.1]

Figure 1 illustrates a simplified view of the Xeon Phi with the MIC architecture. All cores and the RAM are connected over the ring interconnect. There are more components connected even though these are not mentioned.

Figure 1: Overview of the Xeon Phi. Cores are connected over the ring interconnect.

Xeon E5-2699 v3 on the other hand has 18 2.3GHz clocked out-of-order cores and its vector registers (for the AVX-2 instruction set) are only half as big. The Xeon E5-2699 v3 utilizes the new DDR-4 RAM, with a peak bandwidth of 68 GB/s.

Figure 2 illustrates a simplified view of two haswell processors connected via QPI creating one compute node. Each core has private L1 and L2 cache, yet all share the L3 cache. Remote memory accesses from one processor is done over the QPI connection, which has a lower bandwidth then accessing local memory.

2.2 IN-ORDER AND OUT-OF-ORDER CORE

The Xeon Phi cores extends the x86 instruction set and follows an in-order execution paradigm. This means that the scheduling of program instructions is done statically, according to the
Figure 2.: Overview of a Haswell compute node with two processors. A processor has local RAM and multiple cores all share L3 cache. Two processors communicate over QPI.

order defined by the compiler. This can have performance implications as a thread context stalls while carrying out a load instruction.

The Haswell architecture utilizes out-of-order execution, which means that instructions from the same thread context can interleave with data loads from other instructions. This assumes that there is no dependency between the two instructions. Instead of stalling a next instruction, due to a load instruction, it can still be issued to the execution pipeline.

Figure 3 illustrates the Xeon Phi execution pipeline of a single core. It can issue instructions to the pipeline from the same thread context every second cycle. This means that the full utilization of a core requires at least two threads [31], as otherwise only half its processing capabilities is possible. Vector instructions are fed into the vector pipeline or else executed directly in a corresponding execution unit. Each core has two execution pipelines (the U- and V-pipe) that each can execute an instruction every clock cycle, however with a 4 cycle delay for vector instructions [10, Sec. 2.1]. In order to hide the delay many independent vector instructions should be issued into the pipeline. The in-order core architecture will tend to stall hardware threads in order to either fetch data required to do a
certain operation or as depicted on fig. 3 stall in order to refill a prefetch instruction buffer [31]. This can cause a lot of latency, which can be hidden if there are other hardware thread’s instructions in the pipe that can be executed instead [33]. To emphasize: It is important to utilize several hardware threads within a core to saturate the vector pipeline and to hide the latency of CPU stalling.

Figure 3.: Simple illustration of the Xeon Phi execution pipeline.

Figure 4 illustrates the Haswell execution pipeline of a single core. Unlike the Xeon Phi the Haswell execution unit can issue instructions every cycle from the same hardware thread context, which means utilizing hyperthreading technology for very computational intensive applications can cause the performance to degrade, as these can start to compete for the core’s execution units. Decoded instructions are stored in the Reorder Buffer and potentially reorders instructions, while waiting for scheduling of the Scheduler [20, Sec. 2.3.3]. The Scheduler will map decoded instructions to a specific port depicted in fig. 4. Port 0-4 are execution lines connected to computational units and port 4-7 are for memory operations. The Scheduler can issue up to 8 micro-operations (one for each port) every cycle if the Reorder Buffer contains the appropriate decoded instructions.
Port 0 and port 1 are connected to FMA execution units, which allow two SIMD FMA instructions to be executed every clock cycle [20, Sec. 2.1.2]. This results in a throughput of 16 double-precision operations per cycle.

In contrast, the Xeon Phi can only utilize certain instructions in the V-pipe in parallel with executions in the U-pipe. This means that the Xeon Phi can only issue micro-operation into both its execution pipelines if the instructions don’t conflict. This includes vector mask, vector store, vector prefetch and scalar instructions, however not FMA instructions. This implies that the Xeon Phi’s throughput is one FMA instruction per cycle per core.

2.3 Vectorization

Vectorization makes it possible to perform several identical scalar operations with fewer clock cycles then doing the scalar opera-
tions one by one. Each Xeon Phi core has a VPU with 512-bit wide vector registers, which allows one to store 16 single-point or 8 double-point float values in a single register. The Xeon Phi implements the AVX-512 instruction set. Each hardware thread has 32 private entries in the vector register file and can issue VPU instructions that are executed with one CPU cycle throughput, however with a four-cycle delay. The delay can be hidden if the VPU’s execution pipeline is fully saturated [30].

The Haswell processors implements the AVX-2 instruction set that can utilize up to 256 bit wide vector registers, which is equivalent to 8 or 4 single- or double-precision float type values. Similarly as the Xeon Phi there is a latency before a Single Input Multiple Data (SIMD) instruction actually can be utilized. Thus it is critical that the vector pipeline is fully saturated with independent vector instructions, so that throughput becomes one vector instruction per cycle per core.

The vectorization units perform vector loads at the same width of their vector registers and has optimal data movement between L1 D-cache and the vector register if initial data accesses are aligned [22]. Assumed data-dependencies will prevent auto-vectorization, however these compiler assumptions can be enforced or ignored using the correct compiler directives. Vectorization with ignored assumed data-dependency may cause differing results then originally expected. Vectorization requires some attention when it comes to data layout in RAM and how the data is accessed.

```
for(i = 0; i < SIZE; i++)
  a[i] += b[i];
```

![Figure 5.: Higher level example: Normal scalar addition compared to SIMD addition.](image)

In the figures figs. 5 and 7 to 9 we assume that the vector width is 8 doubles wide (512 bit) and that we compare 8 scalar additions with one SIMD-addition. Figure 5 suggests an abstract way of imagining vectorizations compared to normal scalar additions. In fig. 6 and fig. 7 it becomes clear that vectorizing on data with a continuous data access pattern has a clear
for(i = 0; i < SIZE; i++)
a[i] += b[i];

Figure 6.: Higher level example: Normal scalar addition. It implies 8 loads for each element required in the scalar operation and 8 stores back to cache for each individual result.

advantage when it comes to data loads between L1 data cache and CPU register files. Scalar additions requires 16 load instructions for fetching values to do eight summations and 8 store instructions to store results. To compare vector addition requires 2 vector length load instruction and 1 vector length store instruction (this assumes good data access and data aligned access discussed in section 2.3.3). Also the actual additions are done in 8 cycles each for the scalar addition, where the vector length additions can be done in 1 cycle. However, if data access is non-continuous then vectorizing is less efficient.

for(i = 0; i < SIZE; i++)
a[i] += b[i];

Figure 7.: Higher level example: SIMD addition. Only 2 vector loads and one vector store compared to scalar addition in fig. 6.

Figures 8 and 9 suggest that more vector length loads to the vector register files are required for non-continuous data. This is because data must be loaded into the vector register file and then each corresponding value must be extracted into a vector register used for SIMD addition. In the continuous case all data
could be read into one vector register at once and be ready to use. For this kind of non-continuous loads and stores there are special vector instructions called ‘gather’ and ‘scatter’ that are meant to make this process more efficient. Either way if loops are large enough, then vectorizing loops adds a performance increase. If loops are too small, then the delay of vector issued instruction becomes apparent, as mentioned in section 2.2 and scalar operations could potentially be more efficient.

Figure 8.: Higher level example: Non-continuous data makes data fetching for SIMD operations more complex. Data is in this case present in 8 different cachelines instead of 1. Modern processors support some kind of ‘gather’ instructions to handle this more efficiently.

2.3.1 Auto Vectorization

The intel compiler will try to vectorize loops if the ‘-O2’ compiler switch or higher is used during compilation. It can also be activated by including the ‘-vec’ switch. In order to make sure the best vector instructions are used another compiler switch must be added suggesting what instructions to use. For the Xeon E5-2699 v3 the AVX2 instructions are the most appropriate and can be activated with the switch ‘-CORE-AVX2’. For the Xeon Phi only the ‘-CORE-AVX512’ switch can be applied. Better yet, the ‘-xHost’ switch will make sure the most appropriate vector instruction set is used and can then be used for both compilations.

In order to utilize FMAs instructions on a Haswell the ‘-fma’ switch must be included during compilation with an explicit
Higher level example: Non-continuous data makes data storing of a resultant vector more complex. Data should in this case be stored in 8 different cachelines instead of 1. Modern processors support some kind of ‘scatter’ instructions to handle this more efficiently.

Figure 9: Higher level example: Non-continuous data makes data storing of a resultant vector more complex. Data should in this case be stored in 8 different cachelines instead of 1. Modern processors support some kind of ‘scatter’ instructions to handle this more efficiently.

switch defining what vector instructions to use, for example ‘-CORE-AVX2’.

Intel’s compiler icc version 15.0.2 applied AVX-2 and FMA instructions when ‘-xHost’ was included. However older versions, such as version 14.0.4, need to have the mentioned switches explicitly included. The best is to make a trial compilation to assembler code to see that the expected code is generated. A fast glance looking for instructions using the ymm registers indicate the AVX-2 is applied. zmm indicate that the AVX-512 is used. Any instruction containing the substring fma will indicate that FMA instructions are used.

Auto vectorization depends on if there are any data dependencies that might conflict with SIMD operations. If the compiler suspects that there is a data dependency, then it will not vectorize a loop, unless it is being forced through compiler directives. Vectorization can also not be nested, which means that nested loops will only effectively have one loop vectorized.

2.3.2 Data dependencies

Data dependencies in loops occur when these cannot be vectorized without breaking program behavior. These are usually categorized as flow and output dependencies. Assumed data de-
dependencies are when the compiler cannot guarantee that a loop has no dependencies. The compiler is conservative and will skip auto-vectorization due to these assumptions. Assumed flow dependencies happen when there is a so called read-after-write situation illustrated in Code 1. If the pointers ‘a’ and ‘b’ refer to the same memory space, then the vectorization might use values that are not updated in the Right Hand Side (RHS). This alias assumption can be explicitly overridden in three ways. One of them is to use compiler directives ‘#pragma ivdep’ as in Code 2, which tells the compiler to ignore all assumed data dependencies. Another way is to use the ‘restrict’ keyword on variable declaration, which tells the compiler that variables are not aliased, see Code 3. Lastly using the compiler switch ‘-fno-alias’ tells the compiler to simply ignore the alias assumption. The last switch can be dangerous, as it is up to the programmer to make sure that pointers are not aliased.

Explicit flow dependencies must be resolved by the programmer by changing the code using techniques such as loop breaking. If for example in Code 1 ‘b’ would actually be ‘a’, then there is an explicit flow dependency and a loop should not be vectorized.

Output dependency occurs when for example data are written in several iterations. Code 4 writes in every iteration to ‘a[i]’ and then to ‘a[i+1]’, where in total each element is written twice.
in separate iterations. In a vectorized context both writes may not be seen and causes different results. One way to solve this approach is to break up the loop suggested in Code 4 into two.

In order to override any assumed data dependency and just directly vectorize one has to use vectorization enforcements, using the compiler directive ‘#pragma omp simd’ (OpenMP specific) or ‘#pragma simd’ (intel). It forces the compiler to vectorize a loop regardless of its data dependency assumptions or any other inefficiencies. This is a fast directive to enforce vectorization, yet it can cause wrong results if data dependencies actually are present. Most likely a more safe approach is to avoid enforcements and instead fix issues by viewing the optimization report on the decisions made and issues encountered by the compiler.

```c
void func1(double* restrict a, double* restrict b)
{
    // ‘restrict’ keyword, a and b do NOT refer to same memory
    for (i = 0; i < size; i++)
        a[i] += b[i]*b[i-1];
}
```

Code 3: restrict keyword tells compiler that declared variables are not aliased.

```c
void func1(double* restrict a, double* restrict b)
{
    // ‘restrict’ keyword, a and b do NOT refer to same memory
    for (i = 0; i < size; i++)
        a[i] += b[i]*b[i-1];
        // Output dependency, writing to a[i+1] and at next iteration again
        a[i+1] += b[i];
}
```

Code 4: No flow dependency (due to restrict), yet output dependency instead.

2.3.3 Data Alignment

Data alignment is when the data is arranged so that it is placed on an address that is a multiple of a specific alignment size. Typically a size that equals the vector register width is good,
so 64 bytes for the Xeon Phi and 32 bytes for the Xeon E5-2699 v3. The VPU will read from or write to the cache in vector width sizes, which is at 32 or 64 bytes (AVX-2 vs AVX-512) [9]. Aligned data allows for much more efficient I/O than otherwise. Assume for example that the cachelines in fig. 10 are 64 bytes wide and the summation requires 8 doubles (vector width). The first value is though not aligned and the VPU must load extra cachelines in order to get all values required for SIMD operation. As can be seen in the figure, this has implications. Instead of only issuing one vector length loads, two have to be made. Also there is some preparation work to be done, which we briefly discussed in the non-continuous addition case. Clearly aligning the data would have been more beneficial.

```
for(i = 0; i < SIZE; i++)
    a[i] = a[i]+3;
```

Figure 10.: Unaligned data access results in additional stores and loads.

```
// Allocate memory aligned to 64 bytes
double* a = __mm_malloc(sizeof(double)*size, 64);

// 'val1' is determined at runtime
a += val1;

for( i = 0; i < size; i++)
a[val2 + i] = i*a[i]; // 'val2' determined at runtime
```

Code 5: The array ‘a’ is originally aligned to 64 bytes, however the increment of ‘val1’ and access to index ‘val2+i’ may not be.

To make sure that allocated memory is aligned one has to use a memory allocator function that respects the alignment requested. For static memory allocation both intel and GNU gcc uses specifiers to declare data as aligned. For intel ‘_declspec(align(N))’
and GNU gcc ‘_attribute_((__aligned__((N))))’. For dynamic memory allocation the intel compiler has the ‘_mm_malloc(..)’ function and in GNU gcc the POSIX standard ‘posix_memalign(..)’ function can be used. N is the number of bytes to align after. This assures that the allocated memory starts at an address that is a multiple of the alignment size given as an argument.

There is a difference though when it comes to aligned data and aligned data access. If for example the pointer is shifted by an in-runtime decided factor, then it becomes impossible for the compiler to know whether or not an initial access will be aligned in a loop. For this the intel compiler has special hints that assist the compiler. The programmer can give assurances to the compiler that a shifting value implies an aligned access or after a pointer shift one can simply state that a pointer has aligned access. This is done using compiler clauses or, as often referred to in this paper, compiler hints, such as ‘assume(..)’ and ‘assume_aligned(..)’.

```c
double* a = _mm_malloc(sizeof(double)*size, 64);
a += val1;

// Assume that 'a' is always aligned after incrementing
__assume_aligned(a, 64);
for (i = 0; i < size; i++)
a[val2 + i] = i*a[i]; // (a+0) aligned to 64 bytes
```

Code 6: ‘_assume_aligned(..)’ makes sure that the compiler understands that ‘a[i]’ is an aligned access. ‘a[val2+i]’ is cannot be determined as an aligned access.

Code 5 is a typical example where original alignment of data does not matter for aligned access. The pointer ‘a’ is incremented by ‘val1’ steps, which is unknown at compile time. This means at the beginning of the iteration the compiler cannot know if the pointer has aligned data access at the first iteration, it must assume otherwise. A solution is to add a compiler hint suggesting that ‘a’ is indeed aligned, which can be seen in Code 6. Code 7 suggests adding a compiler hint to assure that ‘a[val2 + 0]’ is an aligned access, now all accesses in the loop are aligned.

It is important to only use the compiler hints when it is known that the variables affecting pointer values or its data access are in fact aligned data accesses. If a programmer fails
to do this, the compiler will assume aligned access even though it might not be true for a particular data access. This will cause differentiating results, most likely differing from what actually is expected. The compiler directive ‘#pragma vector aligned’ can be used to propose that the next loop will have all aligned access, this is only recommended if it of course is true.

```c
double* a = _mm_malloc(sizeof(double)*size, 64);
a += val1;

_assume_aligned(a, 64);
// Assume that 'val2' base index is an aligned access
_assume((val2 % 8) == 0);
for (i = 0; i < size; i++)
a[val2 + i] = i*a[i]; // a[val2 + 0] and a[0] has aligned access
```

Code 7: ‘_assume(..)’ makes sure that the compiler understands that ‘val2’ implies an aligned access on an aligned pointer. ‘a[val2+0]’ is an aligned access, so is ’a[0]’, thus the loop can be vectorized with an aligned access.

### 2.4 Streaming Stores

Streaming stores are instructions designed for continuous stream of data generated by SIMD store instructions. Both the Xeon E5-2699 v3 and the Xeon Phi support this instruction. The simple idea is that instead of fetching cachelines into cache and then overwriting them with a SIMD store to simply write these into its respective position in memory directly. This avoids unnecessary cacheline fetches and saves in this sense memory bandwidth [33, Chap. 5]. Streaming stores are activated through the compiler switch ‘-opt-streaming-stores’, where the values auto, none and always can be used to control it. The default is auto, which means that the compiler decides when to use streaming stores. We stress on the point that streaming stores complement SIMD stores, which means that this can only be used in combination with vectorization. Also, the first stores must imply an aligned data access.

Script 1 suggests how to compile a program explicitly using the ‘-opt-streaming-stores’ switch.
Script 1: Either always use streaming stores, let the compiler decide when to use them or turn streaming stores completely off.

Compiler directives can also be used to define arrays or entire loops as nontemporal. This means that the data stored in a loop or an array will not be accessed for a while (meaning we don’t want it to be in cache). ‘#pragma vector nontemporal’ is used to define a loop as nontemporal. See Code 8 for an example.

Code 8: Defining all vector stores in a loop as nontemporal.

The Xeon Phi also features a clevict (cacheline eviction) instruction, which allows the compiler to insert instructions that evict cachelines after they are written with the streaming stores. This means that data that has been read into cache can be evicted after it is written to memory with streaming stores, opening up more space for other cachelines. These are generated by default when a streaming store is generated. The cacheline eviction behaviour can be controlled with the '-opt-streaming-cache-evict' switch [27]. See below example:
2.4.1 Optimization Report

The optimization report are generated for each file compiled when the ‘-opt-report=n’ switch is used during compilation. The value of ‘n’ determines how much should be reported. The value of 5 shows data alignment information, vectorization attempts and gains and information about IPO. It also shows information about where streaming store instructions where placed. The report should be understood as how the compiler currently behaves with the information given during optimizations and its issues should be addressed for optimal performance.

2.5 CACHE HIERARCHY

The Xeon Phi and Xeon E5-2699 v3 have different cache hierarchies that is elaborated in this section.

2.5.1 L1-Cache

On both architectures there are L1 caches, the D-cache and I-cache. Both caches are of size 32KB and are 8-way-associative. A cacheline has the length of 64 bytes and the cache eviction mechanism follows a LRU-like replacement-policy. Cache accesses are 3 cycles and cachelines can be loaded within 1 cycle into a CPU register. For vector instructions this depends on the data accessed see section 2.5.3.
2.5.2 L2-Cache and/or Last Level Cache (LLC)

On the Xeon Phi Knights Corner each core has a unified L2-cache that is 512KB and the data in the L1 cache is inclusive in L2. The L2-cache is the LLC\(^8\) and is shared among all cores. A shared cache means that all cores have a local L2 cache, yet can access the other core’s L2 caches. This means that the Xeon Phi has a collective L2-cache size of up to \(512\text{KB} \times 61 = 31\text{MB}\) of data. If data is missed in the local L2-cache it can be fetched from a remote L2-cache via TDs\(^2\) over the ring interconnect\(^{[10]}\) or from main memory if it is missed in the shared cache. Figure\(^7\) depicts the L2 caches and the TDs connected to the interconnect. 

The Xeon E5-2699 v3 has a 256KB large L2-cache in each core. The L2-cache is inclusive of L1 and only accessible within the core. The LLC\(^4\) is a 45 MB large L3 cache and is shared among all cores.

In both architectures the L2-cache is 8-way associative and the cachelines are of 64 byte size.

2.6 Prefetching

Prefetching of data is fetching data into one of the cache memories before it is missed when a memory access is attempted. The idea is that this will reduce cache misses and in that sense CPU stall latency for the Xeon Phi, as the load instructions will yield more cache hits. This subsection describes at an abstract level what hardware prefetching is and how software prefetching can be utilized.

2.6.1 Hardware Prefetching

Modern processors often include a hardware prefetcher that assumes that programs utilize the data locality principle. The Xeon E5-2699 v3 has one for its L1, L2 and L3 caches (if applicable)\(^{[20]}, \text{Chap. 2.2.5.4}\). The hardware prefetcher utilizes the DCU\(^5\), which prefetches a next cacheline if a previously loaded data block was recently accessed. The instruction prefetcher tries to detect access patterns in the code (a long loop for instance) and prefetches then accordingly instruction cachelines.
into the \textcolor{red}{L1}-cache. The \textcolor{red}{L2}-cache and \textcolor{red}{L3}-cache have both a spatial prefetcher, which always fetches 128 Byte data blocks (meaning two cachelines) on a request \cite{20, Chap. 2.2.5.4}. The Xeon Phi also has a hardware prefetcher for its \textcolor{red}{L2}-cache. It can prefetch up to sixteen data streams \cite{33, Chap. 8, Prefetching}.

Hardware prefetching cannot explicitly be controlled by the programmer, it can however in some cases be turned off \cite{37}.

2.6.2 Software Prefetching

Software prefetching is activated through a compiler switch on the Xeon Phi. This means that the compiler will insert instructions into the code that will prefetch data. The software prefetching can be influenced by the programmer on the Xeon Phi by giving the compiler directives through pragmas or through compiler switches during compilation. On the Xeon E5-2699 v3 this has to be done explicitly through code using \textcolor{red}{intrinsic} functions. Code 9 shows how programmer can influence prefetching on a vectorized loop through \textcolor{red}{software prefetching}. The directive is expressed as follows: '\#pragma prefetch \textit{var:hint:distance}'. 'hint' is either 0 or 1 and means fetching data into \textcolor{red}{L1} from \textcolor{red}{L2} or \textcolor{red}{L2} from RAM. 'distance' has a vector length granularity \cite{6, Chap. 5}.

Code 9 prefetches the array 'a' and 'b' with a distance of 1 into \textcolor{red}{L1} cache, which means the next 8 doubles (Xeon Phi vector register width is 8 doubles, see section 2.3) are prefetched. The same for 'a' in the \textcolor{red}{L2} cache, where the distance is 4 that corresponds to 32 doubles. Array 'b' is not prefetched, due to the \#pragma noprefetch b' directive.

In order to utilize these directives the '-opt-prefetch' compiler switch must be used during compilation or '-O2' switch, which implicitly includes the '-opt-prefetch' switch \cite{29}.

\begin{verbatim}
#pragma prefetch a:0:1
#pragma prefetch a:1:4
#pragma noprefetch b
for ( i = 0; i < size; i++)
a[i] = a[i]*b[i];
\end{verbatim}

Code 9: Assume that loop is vectorized, then prefetching of a and b to \textcolor{red}{L1} and \textcolor{red}{L2} cache is hinted to the compiler, with vector distance 1 and 4 resp. b is not prefetched.
**Software prefetching** can be utilized globally by using the `'-opt-prefetch=n:optional'` switch in combination with `'-opt-prefetch-distance=n1,n2:optional'` switch. \( n1 \) is the same distance specified in Code 9 for \( \text{hint} = 0 \) and \( n2 \) corresponds to \( \text{hint} = 1 \), the distance L1 and L2 cache respectively [26]. Specifying different values for \( 'n' \) in `'-opt-prefetch=n:optional'` determines to what degree the compiler should use software prefetching, where 0 is none and 1-4 are different levels where 4 is the highest (2 is default) [25]. Below is an example of how to compile:

```
icc other-compiler-switches -opt-prefetch=4 -opt-
  → prefetch-distance=1,8 -o a.out main.c
```

Note, the compiler directives will override the configurations set in when compiling with the switches just mentioned.

By allowing the compiler to insert software prefetching instructions one or more extra cacheline sized datablocks, in any cache level, can be prefetched potentially increasing performance as it can avoid cache-misses. The Xeon Phi Knights Corner can utilize both the L1 and L2 cache in order to avoid cache misses. If an iteration only computes on data that utilizes data locality principle then it can be more efficient to prefetch many cachelines instead of just one. Then again, hardware prefetching is per default active and might already be sufficient for the needs of a program.
This section discusses the Burgers model and how it is solved using a finite difference method. Stability and accuracy issues are mentioned in section 3.7.

### 3.1 Burgers Equation Background

Burgers equation is a simplification of the Navier-Stokes equation presented by J.M Burgers in 1939. Burgers dropped the pressure variables and the equation has the definition in eq. (1).

\[
\frac{\partial u}{\partial t} + u \cdot \nabla u = \nu \nabla^2 u
\]

(1)

The Burgers equation is a nonlinear parabolic function and models phenomena that come close to fluid turbulence. One flaw that dismisses the Burgers equation of being a valid model for fluid turbulence is the fact that this nonlinear function can be transformed to a linear function (the heat equation) and thus solved, which was proven with the hopf-cole transformations. Instead, the Burgers equation can be used to test the quality of numerical methods that may be applied on more complex and realistic models (such as the Navies-Stokes Equations). This is the purpose of using this specific model. Equation (1) has a similar form as a convection diffusion Partial Differential Equation (PDE), where the non-linearity is caused by its convection term. The equation consists of the independent time variable and independent spatial variable(s). In terms of one, two or three dimensions the spatial variables will be referred to as x, x-y or x-y-z space, where the x, y, z represent axis of the space. Its variables are defined as follows:

- $u$ is the vector field describing how a ‘point’ is moving at some position in space.
• $\frac{\partial u}{\partial t}$ is the derivative expressing the change of the velocity vector in an infinitesimal change of time.

• $\nabla u$ is the gradient of the vector field, expressing the change of the velocity vector in an infinitesimal change of space.

• $\nabla^2 u$ expresses the change of the acceleration in an infinitesimal change of space.

• $\nu$ is the viscosity that determines the thickness of a fluid, which will act as a dampening factor as $t$ progresses. If $\nu = 0$, then $\nabla^2 u$ vanishes and the Burgers equation is inviscid.

### 3.2 The Fictional Flow Problem

One can imagine that in a space there is some element that flows in some particular way. The true function that expresses how this fluid or gas is moving is modelled according to Burgers equation, yet the function is unknown. So one is left with trying to approximate how the system behaves over time, which can be achieved using finite numerical methods such as [Finite Difference Method (FDM)]. One approach is to divide a continuous space into a bounded space with a finite amount of points, where the distance between points can be either fixed or varying. This is called a discretized space, where each point will have a velocity vector describing how the element moves at the specific point, called a vector field. The illustration in fig. 11 represents a 3D space discretized to mesh of points, where the space has been broken down to a 4x4x4 cube. Each square in fig. 11 represents a vector, describing the flow of a position. Within a bounded finite time interval (from 0-N) one can estimate how the velocity vectors will change over time (with a fixed timestep) according to a points neighbours. Figure 12 shows a more conventional 3D vector-field with arrows describing flow instead. In the next subsections the FDMs used to do the approximations will be introduced, including its shortcomings and advantages.

### 3.3 Finite Difference Method

This section describes FDM from the perspective as if they were explicit, since that will be the theory used in the implementation.
Figure 11.: A 3D mesh, where the cubes illustrate points.

Figure 12.: A vector field, where each point has a certain flow.
A finite difference is an approximation of a partial derivative due to its neighbouring points. For example for the partial derivative of $u$ in time (in one dimension) it can be defined as a forward or as a backwards difference (eq. (2) and eq. (3) respectively):

$$\frac{\partial u(t,x)}{\partial t} = \lim_{h \to 0} \frac{u(t+h,x) - u(t,x)}{h}$$  \hspace{0.5cm} (2)

$$\frac{\partial u(t,x)}{\partial t} = \lim_{h \to 0} \frac{u(t,x) - u(t-h,x)}{h}$$  \hspace{0.5cm} (3)

FDMs will discretize the domain space into a discrete collection of finite points, which can be translated to a one, two or three dimensional mesh. This discretized space will be referred to as the "mesh", "grid" or simply $g$, where $g^{(t)}(i)$ equates to the approximated value of $u(t,x_i)$. The domains are each defined as $D(x_i) = \{x_1, ..., x_{nx}\}$, $D(y_j) = \{y_1, ..., y_{ny}\}$ and $D(z_k) = \{z_1, ..., z_{nz}\}$, where $0 \leq i < nx$, $0 \leq j < ny$ and $0 \leq k < nz$, and each point has a $\Delta x$, $\Delta y$ and $\Delta z$ distance in each dimension from its respective neighbours, so that for example $x_i - x_{i-1} = \Delta x$ and so on. The vectors in the mesh will change over time, which means that the vector components in each point will be approximated in a new timestep using FDM. Equation (4) gives us an estimate using the points in the discretized space for the derivative of $u$ in time as a forward difference approximation.

Note: $\Delta t$ is one timestep and $g^{(t+1)}(i)$ gives the value at point $i$ of the x-axis in the space at time $t + \Delta t$.

$$\frac{\partial u(t,x_i)}{\partial t} \approx \frac{g^{(t+1)}(i) - g^{(t)}(i)}{\Delta t}$$ \hspace{0.5cm} (4)

Equation (5) is a central finite difference, eq. (6) is a forward finite difference and lastly eq. (7) is a backwards finite difference. Compared to eq. (4) these are approximations of the first derivative of $u$ in terms of spatial variables ($x$) and can similarly be approximated using the discretized space. The grid consists of a finite amount of points and thus $g^{(t)}(i+1)$ in this case refers to the approximation of a value at $x_i + \Delta x$.

$$\frac{\partial u(t,x_i)}{\partial x} \approx \frac{g^{(t)}(i+1) - g^{(t)}(i-1)}{2\Delta x}$$  \hspace{0.5cm} (5)
\[ \frac{\partial u(t, x_i)}{\partial x} \approx \frac{g^{(t)}(i + 1) - g^{(t)}(i)}{\Delta x} \quad (6) \]

\[ \frac{\partial u(t, x_i)}{\partial x} \approx \frac{g^{(t)}(i) - g^{(t)}(i - 1)}{\Delta x} \quad (7) \]

The finite differences are derived from the Taylor series, which is illustrated in eq. (8), where \( f \) is some function of \( x \) or any other spatial variable. The Taylor series is infinite and must be truncated in order to be able to approximate a value. This is done by dropping the remainder term \( R^{n+1} \). The order of accuracy is dependent on the order of the truncation error, which is the rate of how fast the truncation error approaches zero as \( \Delta x \to 0 \). [2, Chap. 5.4]

\[ f(x + \Delta x) = f(x) + \Delta x f'(x) + \frac{\Delta x^2 f''(x)}{2!} + ... + \frac{\Delta x^n f^{(n)}(x)}{n!} + R^{n+1} \quad (8) \]

\[ f(x + \Delta x) - f(x) - \Delta x f'(x) = O(\Delta x) \]

Thus: \[ \frac{\delta u(t, x_i)}{\delta x} \approx \frac{g^{(t)}(i + 1) - g^{(t)}(i)}{\Delta x} \quad (9) \]

\[ f(x - \Delta x) - f(x) + \Delta x f'(x) = O(\Delta x) \]

Thus: \[ \frac{\delta u(t, x_i)}{\delta x} \approx \frac{g^{(t)}(i) - g^{(t)}(i - 1)}{\Delta x} \quad (10) \]

In other words the finite differences approximated in eq. (6) and eq. (7) have a truncation error of \( O(\Delta x) \) as the remainder term \( R^2 \) has a \( \Delta x \) term (see eq. (9) and eq. (10)). By combining and manipulating different Taylor series around different points it is possible to express second or really any nth derivative in terms of a finite difference. In our case the second derivative in spatial variables is also needed:

\[ f(x + \Delta x) - 2f(x) + f(x - \Delta x) - \Delta x^2 f''(x) = O(\Delta x^2) \]

Thus: \[ \frac{\delta^2 u(t, x_i)}{\delta^2 x} \approx \frac{g^{(t)}(i + 1) - 2g^{(t)}(i) + g^{(t)}(i - 1)}{\Delta x^2} \quad (11) \]
Central Finite Differences are inherently more precise than Forward or Backward Finite differences, which is due to the terms that cancel out when the Central Difference is derived. This means that when approximating with the same number of points in a backwards scheme compared to a central scheme, the central scheme will have a higher order in its truncation error. [2, Chap. 5.4]

3.3.1 Higher Order Finite Difference Approximations

The suggested finite differences in eq. (9), eq. (10) and eq. (11) can all be approximated using a higher order by including more points to the approximation. Below is the form of a central difference of higher order:

\[
\delta u(t, x_i) \approx \frac{a \cdot g_i^{(t)} + b \cdot g_{i+1}^{(t)} + c \cdot g_i^{(t)} + d \cdot g_{i-1}^{(t)} + e \cdot g_{i-2}^{(t)}}{k \cdot \Delta x}
\]

(12)

Where \(g_{i+2}^{(t)}\) is equivalent to \(g_i^{(t)}(i+2)\) and \(a, b, c, d, e, k\) are constants formed when deriving a higher order equation, which sum are zero as in eq. (9), eq. (10) and eq. (11). Equation (12) suggests a Central Difference with a fourth order accuracy. The coefficients will not be derived in this paper and are instead taken from Fornberg’s paper [1].

As higher order approximations reduces the truncation error it will typically cause the approximation to be more accurate. Consequently the number of computations per point will also increase.

3.4 Explicit and Implicit FDMS

FDMS can be seen as implicit or explicit methods. Explicit methods require values from a timestep \(t\) when approximating a new point in a timestep \(t + 1\). Since values from previous timesteps are usually known the approximation of one point is simply solving for an unknown, which is straightforward. Implicit methods on the other hand require values from a timestep \(t + 1\) and/or \(t\) to approximate a new point in timestep \(t + 1\). In order to solve such a problem it is necessary to solve a system of equations that is computationally more complex, especially since the Burgers equation is nonlinear. All approximations
listed in section 3.3 are explicit examples, where the values of neighboring points used to approximate a new value of a point are from the previous timestep, noted as simply \( t \). Equation (13) on the other hand gives an implicit version of eq. (11), where the neighboring points values are unknown since they are from the timestep that is currently being approximated.

\[
\frac{\delta^2 u(t+1,x_i)}{\delta x^2} \approx g^{(t+1)}(i+1) - 2g^{(t+1)}(i) + g^{(t+1)}(i-1) \quad (13)
\]

### 3.5 Explicit Scheme: FTCS Scheme

The "Forward in Time and Central in Space" is a finite difference scheme that uses a Forward difference method to approximate the time derivatives and a central difference method to approximate the spatial derivatives. It is an explicit scheme, which means that points in a new timestep can be computed directly because all points required for the computations are from the previous timestep. [2, Chap. 10.2]

Substituting the derivative approximations (see eq. (4), eq. (5) and eq. (11)) into Burgers equation (eq. (11)) yields the following:

\[
\frac{\delta u(t,x)}{\delta t} + u(t,x)\frac{\delta u(t,x)}{\delta x} = v \frac{\delta^2 u(t,x)}{\delta x^2} \quad (14)
\]

Where in a Forward in Time Central in Space (FTCS) scheme the \( \frac{\delta u}{\delta t} \) is a forward difference (eq. (4)) and both \( \frac{\delta u}{\delta x} \) and \( \frac{\delta^2 u}{\delta x^2} \) are central differences (eq. (5) and eq. (11)).

\[
\frac{g^{(t+1)}(i) - g^{(t)}(i)}{\Delta t} = -g^t(i) \left( \frac{\delta u(t,x)}{\delta x} \right) + v \frac{\delta^2 u(t,x)}{\delta x^2} \\
g^{(t+1)}(i) = g^{(t)}(i) + \Delta t \left( v \frac{\delta^2 u(t,x)}{\delta x^2} - g^{(t)}(i) \left( \frac{\delta u(t,x)}{\delta x} \right) \right) \quad (15)
\]

Equation (15) describes the one dimensional approach for Burgers regarding Finite Forward Differences with respect to time. One only needs initial conditions (a starting mesh with initial values at time \( t = 0 \)) and a way to approximate the spatial derivatives, which was covered earlier in section 3.3. This can now be translated to a three dimensional case that yields three equations, describing the new velocity vector (consisting
of a x-y-z component) for every new timestep. \( g_x^{(t)}(i) \) now describes the value of the x-component at time \( t \) on the position \( i \). The spatial derivatives are now partial derivatives, where each derivative in space implies three partial derivatives with respect to \( x \), \( y \) and \( z \) dimension.

\[
\begin{align*}
g_x^{(t+1)}(i, j, k) &= g_x^{(t)}(i, j, k) + \\
&v \triangle t \left( \frac{\delta^2 u_x(t, x, y, z)}{\delta x^2} + \frac{\delta^2 u_x(t, x, y, z)}{\delta y^2} + \frac{\delta^2 u_x(t, x, y, z)}{\delta z^2} \right) \\
&- \triangle t g_x^{(t)}(i, j, k) \left( \frac{\delta u_x(t, x, y, z)}{\delta x} \right) \\
&- \triangle t g_y^{(t)}(i, j, k) \left( \frac{\delta u_x(t, x, y, z)}{\delta y} \right) \\
&- \triangle t g_z^{(t)}(i, j, k) \left( \frac{\delta u_x(t, x, y, z)}{\delta z} \right)
\end{align*}
\]

(16)

\[
\begin{align*}
g_y^{(t+1)}(i, j, k) &= g_y^{(t)}(i, j, k) + \\
&v \triangle t \left( \frac{\delta^2 u_y(t, x, y, z)}{\delta x^2} + \frac{\delta^2 u_y(t, x, y, z)}{\delta y^2} + \frac{\delta^2 u_y(t, x, y, z)}{\delta z^2} \right) \\
&- \triangle t g_x^{(t)}(i, j, k) \left( \frac{\delta u_y(t, x, y, z)}{\delta x} \right) \\
&- \triangle t g_y^{(t)}(i, j, k) \left( \frac{\delta u_y(t, x, y, z)}{\delta y} \right) \\
&- \triangle t g_z^{(t)}(i, j, k) \left( \frac{\delta u_y(t, x, y, z)}{\delta z} \right)
\end{align*}
\]

(17)

\[
\begin{align*}
g_z^{(t+1)}(i, j, k) &= g_z^{(t)}(i, j, k) + \\
&v \triangle t \left( \frac{\delta^2 u_z(t, x, y, z)}{\delta x^2} + \frac{\delta^2 u_z(t, x, y, z)}{\delta y^2} + \frac{\delta^2 u_z(t, x, y, z)}{\delta z^2} \right) \\
&- \triangle t g_x^{(t)}(i, j, k) \left( \frac{\delta u_z(t, x, y, z)}{\delta x} \right) \\
&- \triangle t g_y^{(t)}(i, j, k) \left( \frac{\delta u_z(t, x, y, z)}{\delta y} \right) \\
&- \triangle t g_z^{(t)}(i, j, k) \left( \frac{\delta u_z(t, x, y, z)}{\delta z} \right)
\end{align*}
\]

(18)
Equation (12) illustrated a higher order equation with 4th order of accuracy, where 5 constants are used to weight the different values of the points. For a finite difference with 8th order accuracy 9 constants are going to be required for the spatial derivatives (as it requires 9 points). We create two coefficient matrices to make the definition of the approximations more compact, one for the first partial derivative \( C_1 \) and the other for the second partial derivatives \( C_2 \).

\[
C_1 = \begin{bmatrix}
\frac{1}{280} & -\frac{4}{105} & \frac{1}{5} & -\frac{4}{5} & 0 & \frac{4}{105} & -\frac{1}{5} & \frac{4}{280}
\end{bmatrix} \tag{19}
\]

\[
C_2 = \begin{bmatrix}
-\frac{1}{560} & \frac{8}{315} & -\frac{1}{5} & \frac{8}{5} & -\frac{205}{72} & \frac{8}{5} & -\frac{1}{5} & \frac{8}{315} & -\frac{1}{560}
\end{bmatrix} \tag{20}
\]

The first partial derivative is defined in eq. (21). The second partial derivative is defined in eq. (22). Note that \( g_x \) is interchangeable with \( g_y \) or \( g_z \), assuming that the summation index \( l \) is added to the respective \( i, j \) or \( k \). This applies for both eq. (21) and eq. (22).

\[
\frac{\delta u_x(t, x_i, y_j, z_k)}{\delta x} = \sum_{l=-4}^{4} C_1[l + 4] \ast g_x^{(l)}(i + l, j, k) \tag{21}
\]

\[
\frac{\delta^2 u_x(t, x_i, y_j, z_k)}{\delta^2 x} = \sum_{l=-4}^{4} C_2[l + 4] \ast g_x^{(l)}(i + l, j, k) \tag{22}
\]

By substituting eqs. (21) and (22) into eqs. (16) to (18) one has an expression to approximate the vector components for every new timestep.

### 3.5.1 Computational N-point stencils

Equations (16) to (18) require neighbouring points to estimate a new vector component at a specific point in the 3D mesh. If the derivatives are estimated using a central difference to a second order accuracy, then a 7-point stencil describes the points required from the old vector field for the computations of the new value, which can be seen in figs. 13 and 14. For an 8th order accuracy 25 points are required as seen in fig. 15.
Figure 13.: 7-point computational stencil for second order accuracy approximation.

Figure 14.: 3D mesh with 7-point computational stencil (shaded cells).
Figure 15: 25-point computational stencil for eighth order accuracy approximation.
3.6 Stability: Explicit vs Implicit

The numerical stability of a solution is dependent on whether or not it produces a bounded or unbounded solution [2, Chap. 7.6.2]. If a solution is unstable, then the solution may “explode” and values get too large or too small to be computed, either causing an overflow or underflow of the data type used. It is thus desirable to have a stable solution in order to get usable results as an unstable solution will yield wrong approximations.

Typically, explicit schemes require a small timestep in order to be stable. Implicit schemes on the other hand do not have this restriction and reasons to lower the timestep are others [2, Chap. 10.13].

3.6.1 FTCS Stability

The FTCS is a conditionally stable scheme, which means that under certain restrictions the scheme will be stable. According to [2, Chap. 10.10.2] convection-diffusion equations will be stable if the FTCS scheme satisfies the following stability criteria:

\[
\left( \frac{u_x \Delta t}{\Delta x} \right)^2 \leq 2\nu \frac{\Delta t}{\Delta x^2} \leq c^2 \leq 2d \leq 1 \quad (23)
\]

This criteria must hold in all the dimensions, where \(u_x\) and \(\Delta x\) is interchangeable to \(u_y\) or \(u_z\) and \(\Delta y\) or \(\Delta z\) (meaning that every dimension has a stability criteria). Unfortunately, the \(c\) term is dependent on the respective vector component in \(u\) and thus a definite starting condition cannot be predicted that will make a scheme stable (since the vector field \(u\) changes over time).

3.7 Accuracy: Explicit vs Implicit

The accuracy of a numerical scheme can be measured as an absolute error or relative error with respect to the true solution [2, Chap. 1.7.2.1]. Explicit schemes will tend to have low values of both the \(\Delta t\) and the \(\Delta x\) in order to have a stable solution. According to the definition of a first derivative as illustrated in eq. (2) a larger \(\Delta x\) would make the numeric values computed deviate more from the true functions derivative, thus making the approximated solution more inaccurate (as the \(h\) doesn’t tend to
zero it moves away from zero). However, by increasing $\Delta x$ it makes it possible to increase the $\Delta t$ value, since the stability criteria (eq. (23)) might allow that. In that sense, one can tune the explicit method for accuracy or speed (larger timesteps).

Implicit schemes on the other hand are unconditionally stable, which means that it only makes sense to tunes parameters to change the accuracy of the solver. In order to get accurate results the $\Delta t$ often must be close to the $\Delta t$ of explicit schemes solving the same problem. Implicit schemes are computationally more intense then explicit schemes and thus explicit solutions are usually preferred from an accuracy perspective [2, Chap. 10.13].

### 3.7.1 True function

As mentioned in section 3.1 the Burgers equation can be transformed to a solvable linear equation. This was done in [8] for either a 1D, 2D or 3D case. By having a true function one can test a solver for accuracy. The error between the approximations and the true function should in relative terms be fixed, which means that for larger time intervals an approximation’s relative error should not increase compared to shorter approximations.
Modern processors introduce wider `SIMD` instructions as well as `FMA` instructions that become more and more common. The Xeon Phi has more than 60 cores compared to `GPU` s that have thousands. Computing clusters become more and more heterogeneous as accelerators become part of the computing nodes \[7\] in order to boost performance. Compared to `GPGPU` the native code executing on the host machine can also be executed natively on the Xeon Phi. In this sense the Xeon Phi can instead be a host part of an `MPI` runtime environment. Running a program on multiple nodes using MPI in a cluster introduces new problems not present on local machines. Process-process or node-node communication time is inconsistent and must be considered when setting up a pure MPI or hybrid model \[35\].

The HPC systems are moving to multi-socket multi-core shared-memory nodes connected on high-speed interconnects \[35\]. The general trend for `SMP` cluster parallel programming tends towards the usage of `MPI` and `OpenMP` standards \[3,35\]. Hybrid implementations seek to utilize the advantages of both shared memory and message passing parallel programming models. Today’s `SMP` system consist of hundreds or thousands of computing nodes. In order to be able to have programs that perform well on larger clusters, these must be parallelized on different levels. Data level parallelism is the lowest level that demands that `SIMD` instructions are utilized efficiently, determining the computational efficiency of threads. Thread level parallelism demands the efficient use of `OpenMP` worker threads, this is the parallelism within each multi-processor or node. At the cluster level `MPI` is typically used \[3,35\], where hybrid or purely `OpenMP` contexts also occur \[35\]. Intel’s lately discontinued `OpenMP` cluster also showed interesting approach on distributed memory programming with `OpenMP` \[14\].

As an accelerator contender there have been similar work done in `General Purpose computing on Graphical Processing`
Units (GPGPU) front, where the combination of CUDA, interleaved communication and MPI are valid options when it comes to finite difference computations. Other research suggests that the Xeon Phi has good parallelization with bad net performance, making it a worse option for unstructured mesh computations than Graphical Processing Units (GPU). Linear equation problems show promising results on the Xeon Phi, both natively and in a clustered context. Linear equation solving efficiency is interesting, as this is basis of implicit finite difference methods. There have been contributions about structured mesh computations, where FDM strategies show good promise on Xeon Phi cluster. An interesting note is that in most cases the Phi has just been considered as an accelerator and not in its native mode.

Cache optimizations have become one of the major techniques when it comes to performance tuning. Strategies aim to improve locality of reference in order to avoid cache latency (on misses). This includes loop interchanges, loop fusion, loop blocking, data prefetching, array padding or data copying.
IMPLEMENTATION

The implementation is based on the FTCS method presented in section 3.5. The solver will implement one solver module that solves a problem set to the eighth order accuracy for a given timestep.

The computational domain will be defined as: “The domain of points that is going to be approximated in each timestep, which is initially set by the initial condition”.

The boundary domain will be understood as: “The domain of points that is not included in the computational domain, which is initially set and fixed by the dirichlet boundary condition”.

The computational stencils defined in section 3.5.1 consist of 25 points, which means 8 neighbours in every dimension. For simplification, some figures and descriptions might refer to a 7-point stencil context.

The goal of the implementation is to be able efficiently approximate vector fields for parameters set by the user before running the program. The structure of the solver can be divided up into the following points:

1. Create 3D mesh
2. Set initial condition
3. Set dirichlet boundary condition
4. Iterate specified number of timesteps (nt)
   a) Iterate over and compute each point in computational domain
5. Output results

1 The start values for a computational domain.
2 Fixed values on the boundary domain.
Step 4 requires the most attention, since the value of \( nt \) is a variable and can potentially be very large. This requires that the computation should be made as efficiently as possible. To make the computations as accurate as possible double-precision floating-point values will be used, which is an eight byte sized \( double \) type.

This chapter covers how the computation function is implemented, how the data layout is implemented and how it was optimized for performance, in terms of data access, vectorization and parallelization.

Throughout this chapter \( u \) or \( un \) are vector fields and their respective components will be referred to as \( u_x, u_y \) and \( u_z \) or \( un_x, un_y \) and \( un_z \). As in section\(^3\) \( i, j \) and \( k \) are \( x, y \) and \( z \) indices.

### 5.1 Program Execution

This section briefly describes how to execute the solver with different parameters or configuration files.

#### 5.1.1 Command-line Arguments

Initialization can be controlled to some extent by run-time arguments at program execution. Currently, command-line arguments can alter loop blocking parameters if applicable (see section\(^5\) \( 4.6 \)), add MPI configurations if applicable (see section\(^5\) \( 5.5.2 \)) and change what configuration file is loaded, see section\(^5\) \( 5.1.2 \). Script\(^3\) lists some examples of how to run. It is also possible to run with a ‘-help’ argument, which prints an appropriate message of how to execute the current solver.

#### 5.1.2 Configuration file

Per default the solver will always try to load a default file defined in the ‘DEFAULT_CONFIG’ macro value, currently set to ‘sample.conf’. If the default file cannot be found, then it is created in the current working directory and then reloaded. The configuration contains values that determine the size of the computational domain, the distance between each point in each dimension (\( dx, dy \) and \( dz \) respectively) and the number of timesteps to iterate over. The default file can be seen in Script\(^3\).
# Default run that uses configuration file ‘sample.conf’
./solver

# Default run that alters tiling parameters
# (tile config: 128x2x2)
./tile_solver -tx 128 -ty 2 -tz 2

# The same as above
./tile_solver -tx 128 -t 2

# Alter tiling parameters and change config file
./tile_solver -tx 128 -t 2 -c config/big.conf

# Alter MPI process topology
# (tile config: 2x2x2, MPI config: 3x3x3)
./tile_mpi -t 2 -p 3

# Alter MPI process topology
# (MPI config: 3x2x1)
./mpi -px 3 -py 2 -pz 1 --config config/big.conf

# Print help message
./solver --help

Script 2: Execution examples for executing solver with different command-line inputs
# Configurations for Burgers–Solver

# Comments are started by ’#’

# The width in x–axis
nx = 1024

# The width in y–axis
ny = 512

# The width in z–axis
nz = 256

# The number of ’time iterations’ to simulate
nt = 10

# Spatial steps dependent variable
# E.g. dx = ep_x/(nx−1); or dz = ep/(nz−1);
# If ’ep_x’,’ep_y’ or ’ep_z’ is set it
# → overrides any value of ’ep’ for the
# → respective x,y or z dimension.
ep = 8.000000

ep_x =
ep_y =
ep_z =

# Time steps
dt = 0.000100

# Viscosity value
vis = 0.010000

Script 3: Default configuration file.
After the configuration file has been loaded a ‘Domain’ and
‘Steps’ instance is created, which defines the problem-set and
the step values for the current process. Code 10 is the struct
defining the ‘Domain’ instance. The $nx$, $ny$ and $nz$ fields are
the resp. widths in each dimension for the problem-set. The do-
main instance also adds values to the $x_{\text{border\_width}}$, $y_{\text{border\_width}}$
and $z_{\text{border\_width}}$ fields, which determine the size of the boundary domain. In each dimension four neighbours are required
to do a stencil computation, so the border must be a minimum
of four. Along the x-axis the border-width is always set to a
vector width (4 doubles on Xeon E5-2699 v3 and 8 doubles on
Xeon Phi) for data alignment reasons and otherwise it’s also set
to 4. Code 11 is the struct defining the ‘Steps’ instance. The $dt$
field is the size of the timesteps applied during iteration. The $dx$, $dy$ and $dz$ fields define the distance to the next points in the
discretized space. These steps are determined from their resp.
‘ep.x’, ‘ep.y’ and ‘ep.z’ that are specified in a configuration file,
such as Script 3. Below is an example on how to calculate $dx$:

$$dx = \frac{ep_x}{nx - 1}$$

Code 10: Instance defining the domain to solve.

Code 11: Instance defining the distances between points in all three dimen-
sions and the value of one timestep ($dt$).
Before solving (approximating) the problem-set for the given number of timesteps on the computational domain, we must initially set the initial conditions and the dirichlet boundary conditions. Then a solver function can be called, which will start to approximate the values for a new time-steps until it has looped 'nt' times. This is done in a similar manner as in Code 12 where each point in the computational domain is iterated over and set to a value. Afterwards the dirichlet boundary condition is applied on the boundary domain which sets all the outer points to a default value, in this case zero.

The initialization module is declared in the 'init.h' header and defined in 'lib/init.c'. It will also use optimization strategies defined later in section 5.4. In section 5.5.1.3 we elaborate issues regarding a Non-Uniform Memory Access (NUMA) node when it comes to memory accesses. Depending on data access order chosen to write initial values to the mesh it can have implications in terms of memory access latency for strategies suggested in sections 5.4.6 and 5.5.2.6. The Xeon Phi has only one RAM and has uniform memory access.

5.3 UPDATE ALGORITHM

Code 12 shows the basic structure of the code required to make updates at every timestep. In every timestep an update of the computational domain is done, which are the three inner loops. Code 12 uses an inlined function that is specified in Code 13 and which macro 'approx_sum_d' is defined in Code 14. There are two vector fields, one that contains values from the previous iteration and a second vector field to which the updated values will be set. \( u_x, u_y \) and \( u_z \) are vector components from the vector field \( u \) and \( un_x, un_y \) and \( un_z \) are meshes with values from the previous iteration and components from \( un \). After every time iteration these pointers are swapped so that the previous statement is true for the next iteration. Variable \( gCoeffs1 \) contains coefficients for the first partial derivation approximation and variable \( gCoeffs2 \) contains coefficients for the second partial derivation approximation. Both the variables \( gDenom1 \) and \( gDenom2 \) are the common denominators for \( gCoeffs1 \) and \( gCoeffs2 \) respectively. Each approximated partial derivative numerator will be divided by the respective
gDenom1 or gDenom2 term multiplied by a \( dx, dy \) or \( dz \) or \( dx^2, dy^2 \) or \( dz^2 \) (depending on if its a first or second derivative approximation and in respect to which spatial variable). These products are referred to as \( gDenom_dx \) or \( gDenom_d2x \), where the \( x \) is interchangeable to \( y \) or \( z \). All coefficients are precomputed before the solving starts. The only difference for the different point approximations are thus the coefficients and the arguments applied to the macro `approx_sum_d`, where the first argument is the component, the second the index of the point to approximate and the third the stride to the next point. The X-component has as we know a stride of 1, the Y-component a stride of \( y \text{stride} \) and the Z-component a stride of \( z \text{stride} \).

```c
... for( it = 1; it <= nt; it++) {
    ...
    for( k = 0; k < nz; k++)
        for( j = 0; j < ny; j++)
            {
                base_index = z_stride*k + y_stride*j;
                for( i = 0; i < nx; i++)
                    {
                        // Approximate vector fields
                        approx_u(...);
                    }
            }
    // Pointer switch
    tmp = u;
    u = un; // new becomes old
    un = tmp; // old becomes new
}
...
```

**Code 12:** Snippet of the outermost loop iterating over time. Each iteration yields a new timestep `it` where \( t = it \ast \Delta t \). Each time iteration loops over all points and updates these accordingly. See Code 13 for more details regarding `approx_u(..)`

Each component update in `approx_u(..)` call contains 12 multiplications plus \( 9 \ast 6 \) multiplications in the `approx_sum_d` macros. This totals to \( 12 + 54 = 66 \) multiplications. There are 6 additions in each component update, with \( 8 \ast 6 = 48 \) more from the `approx_sum_d` macros that totals to \( 6 + 48 = 54 \) additions. Lastly there are 6 divisions. There are in total \( 66 + 54 + 6 = 126 \) Floating-point Operations (FLOP) for each component update.
inline void approx_u(...) 
{
    int index = base_index + i;
    u_x[index] =
        un_x[index] -
        un_x[index]*dt*
            approx_sum_d(un_x, index, 1)/
            gDenom_dx -
        un_y[index]*dt*
            approx_sum_d(un_x, index, y_stride)/
            gDenom_dy -
        un_z[index]*dt*
            approx_sum_d(un_x, index, z_stride)/
            (gDenom_dz) +
            (vis*dt*approx_sum_d2(un_x, index, 1))/
            (gDenom_d2x) +
            (vis*dt*approx_sum_d2(un_x, index, y_stride))/
            (gDenom_d2y) +
            (vis*dt*approx_sum_d2(un_x, index, z_stride))/
            (gDenom_d2z);
    u_y[index] =
        un_y[index] -
        ...
    u_z[index] =
        un_z[index] -
        ...
}

Code 13: Snippet of the inline function that updates a point in mesh of x, y and z component. See Code 14 for more details regarding approx_sum_d(..) and approx_sum_d2(..)
There are three components that are updated for every computational stencil that require a total of $3 \times (126) = 378$ FLOP. This means that to calculate the FLOP of $nt$ timestep we would use the expression in equation eq. (25). This discards the initial calculations done for the preparation phase of the solver(...) function, which is fixed regardless of number of timesteps simulated. In order to calculate FLOPPing-point Operations Per Second (FLOPs) we will divide the FLOP value with the execution time of the solver, expressed in equation eq. (26). This will become important later on in the performance evaluation.

$$FLOP = 378 \times (nx \times ny \times nz) \times nt$$  \hspace{1cm} (25)$$

$$FLOPs = \frac{FLOP}{time ~ executed}$$  \hspace{1cm} (26)$$

Every call to `approx_sum_d(..)` or `approx_sum_d2(..)` (see Code 14) will make 9 loads from the vector field. Each component update will call `approx_sum_d(..)` and `approx_sum_d2(..)` three times and also load the point’s magnitude in each component. This
equates to $3 + 3 \times 9 + 3 \times 9 = 57$ loads. Thus the amount of reads are $3 \times (57) = 171$ doubles or $171 \times 8 = 1368$ bytes per point update. Writes are simply $3$ doubles or $3 \times 8 = 24$ bytes per point update. Total data traffic (in Bytes) after $it$ iterations can be expressed as in eq. (27).

$$\text{total data traffic} = (1368 + 24) \times nx \times ny \times nz \times nt \quad (27)$$

5.4 Sequential Version

In this section the data layout, data access and sequential optimization strategies used are presented.

5.4.1 Data Layout

The vector fields are simply represented as a struct of three arrays of double-precision float values (8 byte in size), where each vector component is represented by an array of points representing the mesh of points. Visually we want to present an illustration as the 3d-mesh in fig. 16. In the figure the innermost squares represents the points in the computational domain and the darker surrounding squares are points in the boundary domain.

Data points will be accessed through each axis respective index variable, $i$, $j$ and $k$ for x-, y- and the z-axis respectively. Since the space is ordered in only one array we must calculate a unique index from the $i$, $j$ and $k$ indices to represent unique points in the array, which is done using so called axis specific strides. The strides are used in order to access the value of a point in a vector component $x$, $y$ or $z$.

We aligned the space in memory to the x-axis, which means an increment in the index position of the array is an increment in terms of the $i$ value.

A y-stride can be thought of as "jumping to the next row". For example two points $(i,j,k)$ and $(i,j + 1,k)$ are in memory $(j + 1 - j) \times y \_\text{stride} = y \_\text{stride}$ double positions apart. Equation (28) refers to the definition of $y \_\text{stride}$.

$$y \_\text{stride} = 2 \times x \_\text{border} \_\text{width} + nx \quad (28)$$

An increment in $k$ is equivalent as "jumping to the next x-y grid". Similarly, $(i,j,k)$ and $(i,j,k + 1)$ are $(k + 1 - k) \times$
Figure 16: A 3D mesh with boundaries (light shadings) and a computational domain (darker shadings).

Figure 17: Memory layout of a 3D mesh. Data is aligned to x-axis, then y-axis and lastly the z-axis (thus the single stride, y-stride and z-stride).

\[ z_{\text{stride}} = z_{\text{stride}} \text{ positions apart. Equation } (29) \text{ refers to the definition of } z_{\text{stride}}. \]

\[ z_{\text{stride}} = y_{\text{stride}} \times (2 \times y_{\text{border width}} + ny) \quad (29) \]

The data access can be summarized in equation eq. (30) and a more visual illustration of how it is structured in memory is illustrated in fig. 17. Figure 17 illustrates a part of an array in which the mesh in fig. 16 is stored.

\[ \text{value at position}(i, j, k) = \text{mesh}[z_{\text{stride}} \times k + y_{\text{stride}} \times j + i] \quad (30) \]
5.4.2 **Struct of array vs Array of struct**

Generally there are always two ways to structure data in the memory: as a **struct of arrays** or as an **array of structs**. **Struct of array** implies in this implementation that all the components are in separate arrays defining the whole discretized space. This has advantages as we will come to find out later in section 5.4.7 in terms of data alignment, which favours vectorization. **Array of structs** on the other hand do not have this advantage due to its structure.

5.4.3 **Access Pattern**

If we iterate over the array of points over the x-axis when computing the next points, then it is possible to exploit data locality to reduce the cache misses. This means that data cache lines read in from memory can be used for several point computations as long as the access pattern sweeps along the x-axis. So, the motivation in section 5.4.1 of aligning data to the x-axis, then y-axis and lastly the z-axis are for performance reasons. Sweeping over the innermost loop that iterates over x-axis can utilize the data locality principle in the cache lines fetched and experience data cache hits in the next few iterations as the cachelines from the stencil computations overlap. Indeed this would also be true for a **array of struct** implementation mentioned in section 5.4.2, yet due to issues regarding data aligned access, mentioned in section section 5.4.7, this will not be further discussed.

Figure 18 represents a partial L1-cache view after a computation of a point \((i,j,k)\), where all the required cachelines fit into L1 D-cache. A neighbouring point at \((i+1,j,k)\) needs the same cachelines to do its computations and uses overlapping cachelines on its data accesses, which can be seen in fig. 19. We understand that the same is true for the computational stencil centered around \((i+2,j,k)\). There is a reusability of each cacheline, which is illustrated in fig. 21. The blue boxes in figs. 18 and 19 will yield cache hits for 16 computational stencils or 16 computations along x. Figure 21 lists the numbers of values that are useful for a certain number of computations. Each row refers to a computation using a number of values from a cacheline. Similarly we can deduce that the y- and z-strides cacheline boxes in fig. 18 or fig. 19 can generate hits for up to 8 different computational stencils. However, this assumes that
data required for the 16 neighbouring points in x-axis fits into L1 D-cache. Figure 20 gives another illustration on overlapping stencils, where cacheline fetches can be reutilized.

5.4.4 Prefetch potential

As stated in section 2.5.1 both the Xeon E5-2699 v3 and the Xeon Phi has 32 KB L1 D-cache with 64 byte large cachelines. If we count the number of boxes in fig. 18 we see that we need 18 cachelines per component calculations. Also, the updated components will be stored in the second vector field, which needs one cacheline for each component. This totals to 57 cachelines compared to the 512 cachelines available, which equates to 3648 Bytes required in the L1 D-cache. There is space for at least seven more 57 cacheline chunks in the innermost loop over the x-row. For the Xeon E5-2699 v3 we have to rely on the hardware prefetcher, however there is potential room for software prefetching for the Xeon Phi Knights Corner, as mentioned in section 2.6.2. Prefetching seven cachelines is most likely excessive, since we know that the hardware prefetcher mentioned in section 2.6.1 will also have an effect as it prefetches nearby data blocks implicitly. Also there is a higher chance that we thrash the cache by prefetching more cachelines (see section 5.4.5). Regardless, prefetching in L1 and also L2 might prove itself efficient as it can reduce the L1 D-cache miss/hit ratio. However, its ideal parameters must be decided through testing. In any case, Code 15 illustrates one approach where we insert compiler directives telling the compiler to prefetch an extra vector length of data into L1 and two extra into L2.

5.4.5 Cache Trashing Avoidance

Cache trashing occurs when cachelines are fetched and inserted into a cache and lead to cacheline eviction of a cacheline currently in use. This has the effect that a lot of I/O is spent trying to refetch cachelines that are needed for computations. Cache lines are added to a certain block in a cache, determined by the associativity and size of the cache.

The L1 D-cache for example has 32KB of space and is 8-way associative. From this we can derive that this cache has \((32 * 1024) / 64 = 512\) cachelines (cacheline is 64 bytes) and has \(512 / 8 = 64\) blocks, each block holding 8 cachelines. The block to which a cacheline is inserted depends in L1 on its physical
L1 D-Cache: Compute point \((i,j,k)\)

![Figure 18: Very simplified view of a part of L1 D-cache. Data viewed are points from a computational 25-point stencil centered at \((i,j,k)\). All the indices are relative to \((i,j,k)\). Bold text and non-black elements are part of stencil.](image-url)
Figure 19: Very simplified view of a part of L1 D-cache data viewed are points from a computational 25-point stencil centered at (i+1,j,k). The data is already in cache from a previous computation. All the indices are relative to (i,j,k). Bold text and non-black elements are part of stencil.
... for( it = 1; it <= nt; it++)
{
...
for( k = 0; k < nz; k++)
    for( j = 0; j < ny; j++)
    {
        base_index = z_stride*k + y_stride*j;

        // Prefetch 1 vector length data to L1
        #pragma prefetch un:0:1
        // Prefetch 2 vector lengths data to L2
        #pragma prefetch un:1:2
        for( i = 0; i < nx; i++)
        {
            // Approximate vector fields
            approx_u(un,...);
        }
    }
...
...
address and what cache block index it is relative to. Assuming that a cacheline at address ‘a’ is inserted into a block at index 0, then this will be true for every 64th cacheline at some address $a + 4096 \times n$ (4096 is the size of 64 cachelines where ‘n’ is any arbitrary natural number). This means that if there are 9 cachelines needed for several computations and they all have a physical address of the form $a + 4096 \times n$, then these will conflict with each other and cause cache trashing at that particular cacheline block in L1 D-cache.

Another interesting note is that cache trashing is not unique to the L1 cache, yet can also happen in higher level caches in L2 and L3 the address span is just higher because there are more cacheline blocks.

Our technique to avoid this issue is by making sure that the $y\_stride$ and $z\_stride$ values are not divisible by ‘4096/8 = 512’ (every 512th is a conflict) for L1 and ‘32768/8 = 4096’ for L2 in the Xeon E5-2699 v3 and ‘8192’ for Xeon Phi (twice as large cache). This is done through array padding, by increasing the width of the x-dimension, as this implies the least increase in memory, to make the above condition true. The user can also
Figure 21: "Life cycle" of a cacheline that holds neighbouring points in x-axis. Numbers on each line correspond to a relative position to index i, when j and k are fixed. It will yield hits for up to 16 different computational stencils. This life cycle correspond to the life cycle of the blue boxes in fig. 18 and fig. 19.
insert a problem set of which the dimensions a power of 2 (512, 1024,... etc), as these will be padded by a boundary anyways.

5.4.6 Loop Tiling

Loop tiling or loop blocking is an optimization technique used to ensure that data within a working set remains in cache until it is not needed anymore, in order to maximize the positive effects of cache utilization. This typically means that instead of focusing the loops suggested in Code 12 on the entire computational domain, we divide the domain in to subdomains that are updated one by one. This has the benefit of experiencing cache-effects that increase performance with the cost of longer and more complex code. Code 16 is a first example on loop blocking when applying it on Code 12. The three outermost loops are the global iterations, where each step is a tiling width in the resp. dimension. The three innermost loops iterates over a subdomain and does all the computations. We reduced the code by merging the outermost loops into one loop instead. Code 17 illustrates the loop merge, however now a new variable has been introduced called 'num_tiles', which represents the number of subdomains to be computed. The number of subdomains is precomputed, before the time loop in Code 12 starts.

```c
... for ( kk = 0; k < nz; kk += tile_width_z )
    for ( jj = 0; jj < ny; jj += tile_width_y )
        for ( ii = 0; ii < nx; ii += tile_width_x )
            for ( k = kk; k < kk + tile_width_z; k++)
                for ( j = jj; j < jj + tile_width_y; j++)
                    {
                        base_index = z_stride*k + y_stride*j + i;
                        for ( i = 0; i < tile_width_x; i++)
                            {
                                // Approximate vector fields
                                approx_u(...);
                            }
                    }
... 
```

Code 16: Loop tiling with six nested loops.

For larger problem sets good values of tiling dimensions for Haswell appeared to be 128x1x32, 128x2x16 or 128x4x8 (x-y-z
order). This can be motivated since slices of the tile would be
of sizes \(128 \times 8 \times 6 = 6144\) B, \(128 \times 2 \times 8 \times 6 = 12288\) B and
\(128 \times 2 \times 8 \times 6 = 24576\) B that all fit into \(L_1\) cache. The total
tiling sizes result in \(197\) KB that fit into \(L_2\) cache (\(L_2\) is \(256\) KB
for Haswell, see section 2.5.2).

As mentioned in section 2.2, the Xeon Phi most likely will
need to run more then one hardware thread per core to fully uti-
"lize the execution pipes, thus the same tiling dimensions would
make sense for two hardware threads sharing a core, as its \(L_2\) cache is twice as big. However, as it seems the tiling configura-
tion \(1024 \times 1 \times 1\) performs the best, where \(1024\) is the max width
in \(x\)-axis of the problem set used. The implementations per-
forms best when there are at least three or four threads per
core.

```
... for( c = 0; c < num_tiles; c++)
{
    // Start indices are evaluated
    kk = ...;
    jj = ...;
    ii = ...;

    for( k = kk; k < kk + tile_width_z; k++)
        for( j = jj; j < jj + tile_width_y; j++)
        {
            base_index = z_stride*k + y_stride*j + ii;
            for( i = 0; i < tile_width_x; i++)
            {
                // Approximate vector fields
                approx_u( ... ) ;
            }
        }
...```

Code 17: Loop tiling with 4 nested loops.

5.4.7 Auto Vectorization

Keeping the points raised in section 2.3 in mind, auto vecto-
"rization is an easy addition allowing the compiler to choose
the correct vector instructions for the respective platform that code
is compiled for. The compiler requires in most cases compiler
hints or compiler directives in order to vectorize more complex
loops efficiently, as mentioned in section 2.3. This implementation uses compiler hints in favour of compiler directives because the data accesses made are not straightforwardly aligned. Instead, the compiler receives hints that all the pointers are aligned and their respective base index value will lead to an aligned access, where possible. Based on this the compiler may conclude if data access is aligned or not and insert instructions according to information given. Minor tests appeared to show that vectorization enforcement generated slower code when compared to the auto vectorization done by the intel compiler or using the OpenMP directive, which seems to suggest that auto-vectorization includes some efficiency steps not considered in enforced vectorization.

The program is strictly compiled for no alias assumption, as suggested in section 2.3.3. The data alignment hints are placed as can be seen in Code 18. Compiler alignment hints are favoured over the compiler directive '#pragma vector aligned'. All data accesses in the loop are aligned if all the first accesses within a loop are aligned. Viewing Code 14 it becomes clear that if a data access ‘a[base_index + i]’ is aligned, then accesses to neighbouring points at ‘a[base_index + i - 1]’ along ‘x-dimension’ will not be. This is simply because the stride along x-access does not assure aligned access (since it’s 1). Using this analogy, we can predict that the code implies six for the Xeon E5-2699 v3 and up to eight for the Xeon Phi unaligned accesses on the initial calculation for one point. This is also what the generated vectorization report seems to suggest. The other neighbouring calculations using y- and z-stride assure aligned access, as long as the pointer is aligned, through the ‘assume(..)’ function.

5.4.8 Streaming Stores

In Code 13 we see that there is one write made to memory per component update. As mentioned in section 2.4 streaming stores allows us to reduce cache pollution and potentially reduce memory bandwidth. All our writes in the initializer phase (see section 5.2) and in Code 13 can be vectorized and are only used once per iteration over the mesh. So streaming stores seems to be appropriate. We force the usage of streaming stores by simply adding the compiler switch ‘-opt-streaming-stores always’ during compilation of any of our programs.
// Assure that pointers are aligned
_assume_aligned( .. )

// Assure that strides imply aligned access
_assume( ( y_stride \% ALIGNMENT_SIZE/sizeof(double))
\rightarrow \equiv 0 );
_assume( ( z_stride \% ALIGNMENT_SIZE/sizeof(double))
\rightarrow \equiv 0 );

... for( it = 1; it <= nt; it++)
{
    for( c = 0; c < num_tiles; c++)
    {
        // Start indices are evaluated
        kk = ..;
        jj = ..;
        ii = ..;
        ...
        for( k = 0; k < nz; k++)
            for( j = 0; j < ny; j++)
            {
                base_index = z_stride*k + y_stride*j;
                // Base index implies aligned access
_assume( ( base_index \% ALIGNMENT_SIZE/
\rightarrow sizeof(double)) == 0 );
                for( i = 0; i < nx; i++)
                {
                    // Approximate vector fields
approx_u( .. );
                }
            }
    }
}
...

Code 18: Compiler hints place to allow auto-vectorization to vectorize appropriately.
In this section the parallelization strategies on the sequential optimized implementation presented in section 5.4 are presented. This includes shared memory parallelization using OpenMP, message passing parallelization using MPI and the combination of both. OpenMP is activated simply by compiling with 'openmp' switch and Message Passing Interface (MPI) is activated by compiling with a MPI compiler. Also, the MPI implementation expects a 'MPI' macro to be defined, which is done automatically in the makefile of this implementation.

5.5.1 OpenMP

OpenMP (OpenMP) is an API and library for shared memory programming and is a solution supported by the major platforms for C, C++ and Fortran languages. It includes compiler directives (the pragmas), library routines and environment variables that defines run-time behavior. It is portable and a good way to ensure that shared memory parallelization will function on most modern systems, as the host machine/system only requires an OpenMP implementation to be able to compile the desired program. The following OpenMP constructs and directives used are extracted from the OpenMP specification 4.0, as of July 2013.

5.5.1.1 OpenMP Constructs

The parallel region is one of the basic OpenMP constructs used to parallelize blocks of code [13, Chap. 2.5]. It follows the fork-join model, which implies that a master thread that reaches a parallel region will fork to more threads that will do work and later join at the end of the region. Every line of code in a parallel region is executed by the working threads within the parallel context. The number of threads can be controlled in runtime by setting the environment variable 'OMP_NUM_THREADS=N', where 'N' is the number of threads, before program execution or by calling library routines. Code 19 shows an example how to make a 'hello world' program parallel by simply adding one line of code. If one wants to parallelize loops within parallel regions, then a for-construct must be specified. Without a for-construct the for loop will simply be run individually instead of collaboratively.
Loop parallelization is done using OpenMP for-constructs, which is how the distribution of work is done in this implementation. At the end of OpenMP constructs there is an implicit barrier that forces synchronization between all the worker threads. As the number of threads grows the synchronization overhead increases.

The Code 17 can for example be parallelized in the outermost loop, which will consider one tile as the smallest granularity work that can be processed by one thread. Code 20 illustrates how to use a for construct to apply parallelization on a for loop, within a parallel region. An alternative is to leave out the parallel region and instead of writing ‘#pragma omp for’ to write ‘#pragma omp parallel for’, which creates a new parallel context for every new timestep. As creating parallel regions comes at an OpenMP overhead cost, the previous method should be favoured even though the other variant requires less addition of code. It is important that there is no data dependency in the loop as for section 2.3, as there will otherwise be a data race condition.

In order to make sure that some variables are shared with other threads and some are private, it is possible to use OpenMP clauses in OpenMP constructs. The ‘private’ clause will initialize private variables that will become local variables for a worker thread in the parallel region. The ‘shared’ clause marks that certain variables should be shared in the parallel region between worker threads.

```c
#pragma omp parallel
printf("Hello world!\n");
```

Code 19: Parallelization with one OpenMP parallel region

5.5.1.2 Work Scheduling

There are several scheduling methods such as ‘static’, ‘guided’ or ‘dynamic’ that can be applied on runtime using the environment variable ‘OMP_SCHEDULE=scheduling method’ or a library routine call. The scheduling methods are used to divide up the work of a parallelized for loop using a for-construct. ‘static’ will divide the working set into more or less equally sized ‘chunks’ and statically assign the chunks in a round-robin fashion (thread order). The ‘dynamic’ method will hand out ‘chunk-sized’ worksets to a working-thread that requests it. ‘Guided’
// it and the indices variables are private
#pragma omp parallel default(shared) private(it, c, k, j, i, kk, jj, ii)
{
    for (it = 0; it < nt; it += 2)
    {
        // Use a runtime defined scheduling method
        #pragma omp for schedule(runtime)
        for (c = 0; c < num_tiles; c++)
        {
            // Start indices are evaluated
            kk = ...;
            jj = ...;
            ii = ...;

            for (k = kk; k < kk + tile_width_z; k++)
                for (j = jj; j < jj + tile_width_y; j++)
                {
                    base_index = z_stride * k + y_stride * j + ii;
                    for (i = 0; i < tile_width_x; i++)
                    {
                        // Approximate vector fields
                        approx_u(...);
                    }
                }
        } // IMPLICIT BARRIER AFTER FOR
    } // IMPLICIT BARRIER AFTER PARALLEL REGION

Code 20: Parallelization with one OpenMP parallel region and for-constructs.
method is in principle a ‘dynamic’ approach, yet it changes the ‘chunk-size’ depending on the number of iterations left.

Static scheduling is typically a good approach if the iterations contain the same amount of work, which means that each working thread will receive the same amount of work. A ‘dynamic’ or ‘guided’ approach is often more appropriate if it is known that there is a load balancing issue, since these scheduling methods will dynamically hand out work. Dynamic scheduling requires synchronization and adds an additional overhead to parallelization. The scheduling methods require some type of granularity or they will otherwise fallback on a default value. For further information the reader is urged to look at the OpenMP specification \[13\], Chap. 2.7.1.

One consideration was to use static scheduling with a default granularity. The motivation was that in terms of workload the task chunks will be similar, which is true. However, smaller chunk sizes offer the chance that threads working sets overlap, which is illustrated in fig. 23. Assuming that the threads more or less will compute in the same speed, then these will have mutual cachelines fetched from L3 cache. This of course assumes that the threads are on the same processor. Figure 22 illustrates static scheduling with default granularity for two threads on a 2D mesh. Assuming that the cachelines have been replaced after a tilerow, then there will be no possibility of sharing the cachelines with another thread. This is hard to predict and this is why we must talk in terms of chance and possibilities.

Dynamic scheduling is also an option, which in fact provides the best performance on the Xeon Phi, with granularity 1. The Haswell appears to perform equally using either a dynamic or static scheduling with a small granularity.

5.5.1.3 Thread Memory Pinning

On modern NUMA systems memory is usually first allocated when memory blocks are ‘first touched’. This implies that in a continuous virtual memory space the physical memory can be divided to separate RAM’s. In practice memory is always allocated to the closest memory, which implies the local RAM for the specific processor that a thread executes on. A typical ‘malloc(…)’ function will not touch the memory and thus physical allocation is done when the threads starts writing to the memory. \[36\]

This has some interesting effects, as the initialization phase, suggested in section 5.2 will influence the access times to mem-
Figure 22: Static scheduling with default granularity. If unlucky there is no working-set overlapping and mutual cachelines between threads are not present.

Figure 23: Static scheduling, yet with fine granularity (4 in this case). If lucky threads will have overlapping working-sets and experience more cache benefits in L3.
ory for each thread computing on its working-set in the update algorithm. The initialization phase might for example favour a static scheduling with default granularity as its working-set doesn’t overlap with any other threads. We suggest using a finer granularity in section 5.5.1.2 for the compute phase, which can create a situation that a thread accesses remote data allocated by a thread operating on the other processor. Using a finer granularity for the initialization will increase time needed to allocate as the data locality is reduced, however with possible benefits when it comes to computations instead. This means that data access order in the initialization phase has to be the same in the computing phase, meaning that loop tiling suggested in section 5.4.6 should also be applied on the initialization loop together with the appropriate work scheduling strategy.

The ring interconnect on the Xeon Phi allows for a uniform memory access latency regardless which shared memory a core wants to access. In this sense the mix of static work scheduling with default granularity on initialization and work scheduling with a dynamic work scheduling with fine granularity is possible with no side-effects.

For the Haswell we favour a fine granularity for static scheduling for both the initialization and the computation phase, as an increase in initialization time is preferred over potential increase in latencies when it comes to data memory accesses.

5.5.2 MPI

MPI is a specification for a message-passing system that is used widely. There are different implementations, such as for example intelmpi or openmpi. This section covers message passing concepts and MPI specific functions used in order to parallelize the implementation using MPI. Processes in a MPI context will be referred to as MPI processes. Most of the code additions for MPI are declared in ‘headers/mpi-lib.h’ and defined in ‘src/lib/mpi-lib.c’, yet some code-blocks are activated in the code base if the ‘MPI’ macro value is defined.

5.5.2.1 MPI Concepts

The MPI concept is that instead of having threads collaborating and communicating over shared variables, several processes will communicate by the means of messages. The computa-
tional domain is broken down into several smaller domains, where each domain is dependent on its neighbouring domains (or processes). In order to update all points within a computational subdomain an area of ghost cells is introduced. Ghost cells are a group of points that contain points used to update the overlapping area. An overlapping area is the group of cells that other subdomains are dependent on. Ghost cells do not really exist and are copies of the overlapping area of neighbouring processes. At every new timestep MPI-processes will send their overlapping area to their neighbouring processes. The overlapping area of a MPI-process is sent over to a neighbouring process and stored into the ghost cells. This communication is called the boundary exchange.

Figure 24 illustrates how a computational domain can for example be split into four equal sized sub-domains. Each sub-domain is bound to a process and requires MPI communication before every timestep update. Figure 25 illustrates a boundary exchange at specific X-Y 2D slices. Figure 26 illustrates a boundary exchange for x-axis boundaries in 3D context. The see-through cells are ghost cells and the dark shaded cells is the overlapping area.

5.5.2.2 MPI Virtual Topology

Virtual topologies can be used to represent communication patterns between MPI-processes. There are two methods according to the MPI specification, one of them is as to create a general graph and the other to create a cartesian structure [17, Chap. 7.2].

Naturally, a cartesian structure was applied in this implementation using the ‘MPI_Cart_create(..)’ function. MPI processes
Figure 25.: Boundary exchange at X-Y 2D slice. Overlapping area is sent to a receiving part that copies it to its ghost cell area.

Figure 26.: Boundary exchange between processes in 3D. Overlapping area is sent to a receiving part, which copies it to its ghost cell area.
Figure 27: A one-dimensional virtual MPI_Cart topology. Processes are structured as a cartesian structure. Arrows indicate communication partners.

that are neighbours according to the virtual topology will be interpreted as neighbours when it comes to computational domain splitting. This is illustrated in fig. 27 where each box represents a MPI process and each neighbouring box represents a neighbouring process. The arrows indicate that there will be MPI communication (boundary exchanges) between the processes. Figure 28 is a 2D example illustrating the same. 3D topologies are also possible and sometimes needed but will not be illustrated. One strategy is probably to reduce the number of dimensions for the virtual topology, as it reduces the communications needed per processes.

Except for creating convenient ways to setup a structured communication pattern between MPI-processes, virtual topologies can also be used by the runtime system when it comes to physical mapping on processor cores [17, Chap. 7.1]. The goal is to optimize the communication between processes, which usually means making sure that processes that are neighbours in the virtual topology should receive a fast and preferably high bandwidth communication channel. For instance, a socket-to-socket communication should be preferred over a node-to-node communication. The current MPI specification describes this possibility, yet does not require this feature in an implementation and communication optimization will have to be done manually.

The implementation extends the domain instance specified earlier in Code 10 in section 5.1.2. Code 21 contains a struct that has been added to the 'Domain' struct if MPI is active. It contains values to keep track of its rank in the 'MPI_comm' instance comm_cart that is associated with the virtual topology. It also contains the coordinates in the cartesian topology structure in the coords field and some fields that list the number of processes in every dimension in respect to the topology and what neighbours a current MPI-process has. The value of '-1' typically means that there is no neighbour.
Figure 28: A 2D virtual MPI_Cart topology. Processes are structured as a cartesian structure. Arrows indicate communication partners.

```c
typedef struct {
    int rank;
    int rank_world;
    int size_world;
    MPI_Comm comm_cart;
    int coords[3];
    int processes_in_x;
    int processes_in_y;
    int processes_in_z;
    int neighbour_left_x;
    int neighbour_right_x;
    int neighbour_up_y;
    int neighbour_down_y;
    int neighbour_up_z;
    int neighbour_down_z;
} MPI_Params;
```

Code 21: MPI_Params struct containing MPI related information needed in runtime by the solver.
5.5.2.3  **MPI Point-to-Point Communication**

The basic sending and receiving of messages by processes is the basics of MPI communication [17], Chap. 3.1. Point-to-Point communication is done between two MPI-processes, where one is receiving and the other is sending. This can happen in a blocking or a non-blocking manner. Blocking communication is done through calling 'MPI_Send(..)' or 'MPI_Recv(..)', which will block for the resp. process until the send or receive transmission is done (this has different meanings depending on communication modes). In both cases a buffer is always specified, where a sending buffer is the data to be sent and a receiving buffer is where to receive data. A return implies that respective buffers can either be modified or read. Nonblocking communication can also be utilized through the 'MPI_isend(..)' or 'MPI_irecv(..)' calls. Nonblocking should not be mixed up with asynchronous communication, as it is implementation and platform dependent on how the non-blocking implementation will work. A return from a nonblocking call will not imply that the resp. buffers are ready to be modified or read and should be used with care.

According to the MPI specification there are four communication modes that can be applied for the blocking 'MPI_Send(..)' call. The 'buffered mode' call copies the data into a local buffer (the send buffer) and then it returns. As the data was copied it can be modified on the sender side directly. The *synchronous mode* returns when a receiving end has started to receive the data being sent. The *standard mode* applies either a similar strategy as the *buffered mode* or it completely transfers the data to the receiving end (assuming it is already invoked, otherwise it waits). The *ready mode* only sends data if a receive call was initiated before the send call, if data is sent it will use the same approach as in *standard mode*. The *buffered mode* is in this sense purely local as it only depends on the send data to be copied into a separate buffer. The *synchronous mode*, *standard mode* and *ready mode* are non-local as these may depend on whether or not the receiving end invoked a receive call [17], Chap. 3.4. The *standard mode* behavior depends largely on the MPI implementation, yet it can also be as simple to assume that if MPI cannot allocate a large enough buffer to copy the send buffer, then it will most likely transfer the data directly and block until it is delivered. 'MPI_Send(..)' uses the standard communication mode and is considered throughout the rest of the paper. There is only one blocking receive call, 'MPI_Recv(..)', which returns
when data has been received in the receiving buffer [17, Chap. 3.4].

Nonblocking sends have the same four communication modes as the blocking functions. Nonblocking functions normally initiate a communication and then returns an opaque 'Request' object as reference of the communication. If there is suitable hardware, then the non-blocking communication can be executed concurrently with other operations. Using the 'Request' instance, we can check if a nonblocking communication has finished using 'MPI_Test(..)' or wait until it is done using 'MPI_Wait(..)' call.

This implementation uses only nonblocking sends and receives to exchange boundaries. The motivation is simply because this allowed for code reuse for a computation interleaving version. The suggested implementation of MPI will be given after section 5.5.2.4, which will list the importance of using derived types for MPI communication on non-continuous data accesses.

5.5.2.4 MPI Derived Types

When data is continuous in memory it is very easy to send or receive data using MPI communication. In principle we only specify the start of the buffer and how much should be sent or received. Then, when communication is done the data has been transferred.

There are however cases where non-continuous data blocks need to be transferred. These have to be manually packed into a continuous buffer before they are supplied to the 'MPI_Send' function. As mentioned in section 5.5.2.3 this can imply a copy to a sending buffer. If considering the receiving end, where data also has to be unpacked, it will result into communication following the illustration of fig. 29. Figure 29 illustrates a flow where data has to be packed and unpacked by the programmer, which is tedious and in fact inefficient as it adds more I/O. Figure 30 illustrates the flow of MPI communication with non-continuous data when using MPI derived data types. The excessive copying due to packing and unpacking are now gone, as the MPI derived data types specify to the MPI functions on how to access the non-continuous data.

The solver implementation wants to exchange boundaries along the x-, y- and z-dimension, depending on where in the virtual topology its neighbours are. In this sense, we need one MPI derived data type for every dimension describing the data
access of boundaries. This totals to three MPI derived data types. The MPI derived datatype for boundary exchange over x-dimension is declared in Code \[\text{22}\]. Important to note that data access for overlapping area and ghost cells are always the same, which is why the same MPI derived datatype can be used for both sending and receiving data. Code \[\text{22}\] first defines a MPI derived data type of an overlapping area in a X-Y 2D Slice of a 3D mesh as ‘x_access_type’. This is because the different blocks are separated with the same stride. Next another MPI derived data type is defined based on the previously defined ‘x_access_type’, which separates the different slices by a specific stride. Now the entire 3D mesh has an access pattern on how to access the overlapping area over the x-dimension that MPI can understand, defined as ‘x_transfer_type’.

The same logic applies for the y- and z-dimension overlapping area, that is displayed in Codes \[\text{23}\] and \[\text{24}\].

5.5.2.5 Boundary Exchange

As mentioned in section \[\text{5.5.2.3}\] nonblocking communication is favoured in combination with MPI derived data types, as mentioned in section \[\text{5.5.2.4}\]. The functions doing the boundary exchange are defined in ‘src/lib/mpi-lib.c’ and are called before every update. Code \[\text{25}\] is the implementation with MPI. It is an extension of Code \[\text{17}\] from the loop tiling section, where both ‘START.Exchange_BOUNDARY(…)’ and ‘WAIT.FOR_MPI_COMMUNICATION’ are expandable macros defined in Codes \[\text{26}\] and \[\text{27}\].

Code \[\text{26}\] calls ‘recv_boundaries(…)’ and ‘send_boundaries(…)’ function specified in the ‘mpi-lib’ module. These will start non-
Code 22: Declaration, definition and commit of MPI Derived Datatypes.

```c
MPI_Datatype x_access_type;
MPI_Datatype x_transfer_type;

// Overlapping area is 'x_border_width' wide at 'nx' places in a x-y 2D slice
MPI_Type_vector(ny, x.Border_width, y.stride, MPI_DOUBLE, &x_access_type);

// There are nz slices, so extend the 'x_access_type'
MPI_Type_hvector(nz, 1, 
  sizeof(double)*z.stride, 
  x_access_type, &x_transfer_type);

// Both data types must be committed to the MPI context
MPI_Type_commit(&x_access_type);
MPI_Type_commit(&x_transfer_type);
```

Code 23
MPI_Datatype z_access_type;
MPI_Datatype z_transfer_type;

// Copy entire computational x−y 2D slice
MPI_Type_vector(ny, nx, y_stride, MPI_DOUBLE, &z_access_type);

// Copy ‘z_border_width’ amount of slices
MPI_Type_hvector(z_border_width, 1, sizeof(double), z_stride, z_access_type, &z_transfer_type);

// Both data types must be committed to the MPI context
MPI_Type_commit(&z_access_type);
MPI_Type_commit(&z_transfer_type);

Code 24

blocking send and receives for each component using the respective MPI derived datatype. Each component ‘send_boundaries(..)’ or ‘recv_boundaries(..)’ triggers six calls to ‘send_boundary(..)’ or ‘recv_boundary(..)’, which are the basic functions calling the nonblocking communication functions, see Codes 28 and 29.

The MPI Request instances are stored in the recv_requests or send_requests buffers and later used in the Code 27 where ‘wait_for_io(..)’ is called, which also is part of the ‘mpi-lib’ module. ‘wait_for_io(..)’ simply waits for all MPI Request objects in the respective buffers to finish using the ‘MPI_Waitall(..)’ call, in order to confirm that MPI communication is finished.

5.5.2.6 Interleaving MPI Communication with Computation

As mentioned in section 5.5.2.3, we can with the appropriate hardware and software support interleave MPI communication with computations. We already use non-blocking communication in our MPI implementation suggested in Code 25. If we want to interleave the MPI communication we will have to postpone the ‘wait’ for communications to finish, this is best done after the computations are done. However, we have to make sure that we do not update the overlapping area that is currently being transferred as the ghost cells might not have been received yet and the send transfers might not be done. Figure 31 illustrates the different areas that a programmer will have to consider. The innermost area are the computable cells before MPI communications are done. The overlapping area
for( it = 0; it < nt; it += 2)
{
  ...
  START_EXCHANGE_BOUNDARIES(un_x, un_y, un_z);
  WAIT_FOR_MPI_COMMUNICATION;
  for( c = 0; c < num_tiles; c++)
  {
    // Start indices are evaluated
    kk = ..;
    jj = ..;
    ii = ..;

    for( k = kk; k < kk + tile_width_z; k++)
      for( j = jj; j < jj + tile_width_y; j++)
      {
        base_index = z_stride*k + y_stride*j + ii;
        for( i = 0; i < tile_width_x; i++)
        {
          // Approximate vector fields
          approx_u(...);
        }
      }
  }
  // Pointer switch
  ...
}
# define EXCHANGE_BOUNDARIES(un_x, un_y, un_z) \\ recv_boundaries(my_domain, recv_requests_x, \\
  -> x_transfer_type, y_transfer_type, \\
  -> z_transfer_type, un_x, y_stride, \\
  -> z_stride); \\ recv_boundaries(my_domain, recv_requests_y, \\
  -> x_transfer_type, y_transfer_type, \\
  -> z_transfer_type, un_y, y_stride, \\
  -> z_stride); \\ recv_boundaries(my_domain, recv_requests_z, \\
  -> x_transfer_type, y_transfer_type, \\
  -> z_transfer_type, un_z, y_stride, \\
  -> z_stride); \\ send_boundaries(my_domain, send_requests_x, \\
  -> x_transfer_type, y_transfer_type, \\
  -> z_transfer_type, un_x, y_stride, \\
  -> z_stride); \\ send_boundaries(my_domain, send_requests_y, \\
  -> x_transfer_type, y_transfer_type, \\
  -> z_transfer_type, un_y, y_stride, \\
  -> z_stride); \\ send_boundaries(my_domain, send_requests_z, \\
  -> x_transfer_type, y_transfer_type, \\
  -> z_transfer_type, un_z, y_stride, \\
  -> z_stride);

Code 26: Macro for boundary exchange. Start receive communication and then the send communications.

#define WAIT_FOR_MPICOMMUNICATION wait_for_io( \\
  -> send_requests_x, recv_requests_x); \\
wait_for_io(send_requests_y, recv_requests_y); \\
wait_for_io(send_requests_z, recv_requests_z);

Code 27: Macro for waiting for MPI communication to finish.
```c
void send_boundary(int target, MPI_Params* mpi_params,
                   MPI_Request* request, MPI_Datatype datatype,
                   double* dataptr)
{
    if (target < 0)
    {
        *request = MPI_REQUEST_NULL;
        return;
    }

    MPI_Isend(dataptr, 1, datatype, target, 1,
              mpi_params->comm_cart, request);
}
```

Code 28: Function wrapper of MPI_Isend(...) that sends data, which read access pattern is defined by the supplied MPI_Datatype

```c
void recv_boundary(int source, MPI_Params* mpi_params,
                    MPI_Request* request, MPI_Datatype datatype,
                    double* dataptr)
{
    if (source < 0)
    {
        *request = MPI_REQUEST_NULL;
        return;
    }

    MPI_Irecv(dataptr, 1, datatype, source, 1,
              mpi_params->comm_cart, request);
}
```

Code 29: Function wrapper of MPI_Irecv(...) that receives data, which write access pattern is defined by the supplied MPI_Datatype
can be computed as soon as the communication finishes. So interleaved communication requires two passes over the mesh, one being a pass over the computable cells and the other over the overlapping cells. This is illustrated in Code 30.

Notice that we iterate over the mesh in a similar manner as suggested in Code 17, as we want to emphasize local memory accesses as often as possible as suggested in section 5.5.1.3. This can have implications as we have to check whether a tile is relevant or not for the overlapping area update. This becomes a trade-off between remote RAM access vs code complexity in the second pass. Unfortunately we did not test another strategy for interleaving, leaving room for possible improvements.

5.5.3 Hybrid Implementation

In our case a hybrid implementation is the combination of both MPI and OpenMP. This allows MPI processes to utilize a group of worker threads to compute its sub-domain. Large numbers of worker threads in OpenMP often implies extra OpenMP overhead as worker threads usually synchronize a lot. Running smaller thread groups per MPI process could thus in theory give us a performance advantage, assuming that we can hide MPI overhead. If an interleaved implementation, as suggested in Code 30, is able to hide MPI overhead, then it has potential to perform better than pure OpenMP.
... for( it = 0; it < nt; it += 2) {
 ... 
 START_EXCHANGE_BOUNDARIES(un_x, un_y, un_z);
 for( c = 0; c < num_tiles; c++)
 {
  // normal pass over mesh
  ...
 }
 // Boundary exchange interleaved with computations.
  \rightarrow Wait for communication
 WAIT_FOR_MPICOMMUNICATION;
 for( c = 0; c < num_tiles; c++)
 {
  kk = ..;  // Calculate indices
  jj = ..;
  ii = ..;

  // Evaluate whether tile borders to a ghost area
  if( ghost-area-border )
  {
   // Update overlapping area according to where
    \rightarrow it borders (x, y or z)
    ...
  }
 }
 // Pointer switch
  ...
 }

Code 30: MPI implementation. See Codes 26 and 27 for expansion of macros.
Until this point we have discussed how to utilize OpenMP and MPI in our implementation. In order to utilize a hybrid implementation both strategies just have to be active and a certain number of threads have to be dedicated per MPI process at runtime.
BENCHMARK

This chapter discusses how the solver implementation was benchmarked. The test platforms are introduced and the problem sets used are defined. This is followed by a compilation section, which lists different configurations and compiler switches used for each optimization type. The significance of thread and process pinning is shortly discussed. Lastly speedup graphs and the roofline model are introduced as tools for performance analysis.

In short a solver implementation is executed three times and its fastest time result is noted down. If the implementation supports parallelization it will be run with different number of threads, processes or core and threads. The latter is only applicable for the Xeon Phi executions. The results are summarized in speedup and efficiency graphs mentioned in section 6.8 and presented in chapter 7. The results will be related to the roofline model specified in section 6.9 in the discussion.

6.1 TEST PLATFORMS

There are three systems to consider:

- One node of two Xeon E5-2699 v3 processors, referred to as hostmachine or host.

- One Xeon Phi card, parallelized with OpenMP

- Cray XC40 system, Beskow cluster at PDC in Stockholm. One compute node contains two E5-2698 v3 processors. This node has very similar specifications as the hostmachine. Will be referred to as Cray or Beskow.

The Cray will use up to 128 nodes to illustrate speedup gain and efficiency on the cluster.
6.2 PROBLEM SET

The problem set used for comparing the Xeon Phi with the host is the default problem set defined in the ‘config’ module, discussed in section 5.1.2 in Script 3, referred to as problem-small. This problem size is about 6.44GB large.

On the Beskow cluster a problem set with configurations 1024\times1024\times1024 is also considered and referred to as problem-big. Another problem-set is of dimensions 2048\times2048\times2048 and is referred to as problem-huge. These are of sizes 52GB and 412GB.

6.3 COMPILATION

The target platform is on a linux or unix environment supporting the build utility 'Makefile' and the Intel compiler. The makefile for this project is very comprehensive and all the different optimizations can be activated through a resp. building target in the makefile. Unfortunately the MPI environment was not set up in time on the Xeon Phi’s, so these will not be used for benchmarking with MPI.

Next follows the different targets built for testing and their special flags if any. Vectorization is implicitly used and otherwise stated as comment.

6.3.1 Sequential

All the sequential implementations will compile using the `-O2', `-opt-streaming-stores always' and `-fno-alias' switch.

6.3.1.1 Haswell Architecture

Sequential solver using loop-tiling:

```
make tile_solver
```

6.3.1.2 MIC Architecture

The `-opt-streaming-clevict=3' switch is also used.

Sequential solver using loop-tiling:

```
make mic_tile_solver
```

Sequential solver using loop-tiling and prefetching:
6.3.2 OpenMP

All the OpenMP implementations will compile using the `-openmp`, `-O2`, `-opt-streaming-stores always` and `-fno-alias` switch.

6.3.2.1 Haswell Architecture

OpenMP solver using loop-tiling:

```
make_tile_openmp_solver
```

6.3.2.2 MIC Architecture

The `-opt-streaming-clevict=3` switch is also used.

OpenMP solver using loop-tiling:

```
make_mic_tile_openmp_solver
```

OpenMP solver using loop-tiling and prefetching:

```
export "BURGERS_SOLVER_L1_PREFETCH_VALUE" = 1
export "BURGERS_SOLVER_L2_PREFETCH_VALUE" = 2
make_mic_prefetch_tile_openmp_solver
```

6.3.3 MPI

All the MPI implementations will be compiled using the `-O2`, `-opt-streaming-stores always`, `-fno-alias` and a macro value definition using `-DMPI` switch. `-DMPI` will activate certain blocks of code within the implementation that are MPI specific.
6.3.3.1 Haswell Architecture

MPI solver using loop-tiling:

```
make tile_mpi
```

MPI solver using loop-tiling and openmp:

```
make tile_openmp_mpi
```

MPI solver using loop-tiling, openmp and interleaved communication with computations:

```
make tile_openmp_interleave_mpi
```

6.4 Runtime Configurations

The loop tiling optimization depends on runtime input. So does the virtual topology setup for the MPI implementations, in order to define neighbours for processes.

As mentioned in section 5.4.6, 128x2x16 was a good tiling configuration for the Haswell processor. This is therefore used in the benchmark. For the Xeon Phi the configuration 1024x1x1 was suggested and is therefore used. For further details see section 5.4.6 were the configurations were suggested.

Using other tiling configurations fitting into L2-cache performed worse then the suggested dimensions by a factor of 1.5 or more. This illustrated that the tiling configurations were very important.

The virtual topology discussed in section 5.5.2.2 requires dimensions to divide a group of processes into a 3D process grid. As mentioned fewer dimensions are to be preferred as this reduces the amount of communications made. However, section 5.5.2.2 also discusses that using interleaved communications with computations the subdomains may not be smaller then 8, which must be considered when choosing dimensions.

The approach used for executing the MPI implementations was to split first over the x-axis until the splitting results into a subdomain dimension size that is less then the tiling dimension. Then the y-axis is split and lastly the z-axis.

See Script 4 for some examples.

6.5 Timing

The current implementation solver contains a stopwatch function, which will take the time for initialization and solving of
# Openmp on Haswell
bin/tile_openmp_solver -tx 128 -ty 2 -tz 16

# MPI on Haswell
mpiexec -n 32 bin/tile_mpi -tx 128 -ty 2 -
   – tz 16 –px 8 –py 4 –pz 1

# MPI on Haswell
mpiexec –n 4 bin/tile_mpi –tx 128 –ty 2 –tz
   – 16 –px 4 –p 1

# MPI on Haswell
mpiexec –n 1024 bin/tile_mpi –tx 128 –ty 2
   – –tz 16 –px 8 –py 64 –p 2

# Openmp on Xeon Phi
bin/mic_tile_openmp_solver –tx 1024 –t 1

Script 4: Different ways to execute different configurations of implementation.

the problem-set and print it to the standard output. In a sequential context the ‘gettimeofday(..)’ of the ‘sys/time.h’ system header is used. In a OpenMP context the ‘omp_get_wtime(..)’ of the OpenMP library is used.

6.6 THREAD AND PROCESS CONTROL

Thread creation can be controlled in runtime by setting the 'OMP_NUM_THREADS' environment variable to the number of threads. MPI process creation is dependent on how the MPI runtime is started and does not require a recompilation of a program. The pinning of threads and processes is necessary for two reasons. One is to avoid undesired process/thread migration, which can be issued by the operating system. The other is to minimize NUMA effects. If a thread is moved to another core on another socket or node, then it will not have the same memory access time as before, which can increases memory latency. It is also desirable that threads that share data are grouped together on a socket or node to minimize memory latency.
Increased latency on memory access was for example a reason why the sequential implementation was not used as a reference time, see section 6.7.

6.6.1 Thread pinning

In the intel openmp implementation the setting of ‘KMP_AFFINITY’ environment variable before program execution will control thread pinning. The runs made in the benchmark will use a scattered pinning approach by setting the variable to ‘scatter’. This will place out threads on a core granularity, ensuring that each core has only one thread (assuming there is no oversubscription of threads).

The Xeon Phi has ‘KMP_PLACE_THREADS’ environment variable, which will define how thread placements is done on cores. ‘KMP_PLACE_THREADS=4c2t’ will for example place 2 threads on 4 cores, assuming that ‘OMP_NUM_THREADS’ is set to at least 8. How the threads are placed out is again controlled using the ‘KMP_AFFINITY’ environment variable, using ‘scatter’ or ‘compact’ as values.

6.6.2 Process pinning

Pinning of MPI processes is different depending on in what environment the executable is started. Intelmpi was used with ‘mpiexec’ on the host node and on the Beskow cluster the ‘Application Level Placement Scheduler, ‘aprun’, was used.

6.6.2.1 Pure Intelmpi Process Pinning

The intelmpi implementation uses the ‘I_MPI_PIN_PROCESSOR_LIST’ for pure MPI implementations. This environment variable determines which processor subset a process should be constrained to. The different values are ‘all’, ‘allcore’ and ‘allsocks’, which bind a process to a logical core on a node, a physical core on a node or to a socket on a node. The processor set can be complemented by a mapping value, determining in what way processes are pinned to one unit in the processor subset. ‘bunch’ maps processes close to eachother within a socket. ‘spread’ maps processes consecutively to close units with the possibility of avoiding shared resources. ‘scatter’ will map processes in order to avoid shared resources [15].
There were minor differences using ‘allcores’ or ‘allsocks’. Setting the environment variable to ‘all’ doubled the execution time, which is to be expected as more than one process can be executed per core. The ‘scatter’ mapping value is preferred, which makes sure that processes are mapped evenly on both sockets. ‘allsocks’ is also preferred over ‘allcores’ as it disallows migration of processes to a different socket. I_MPI_PIN_CELL should also be set to ‘core’, whichpins processes to physical and not logical cores. MPI implementations on the host were started as below:

```
# Pure MPI execution on Cray
#
I_MPI_PIN_CELL=core
  ➞ I_MPI_PIN_PROCESSOR_LIST=allsocks :
  ➞ scatter mpiexec -n 32 bin/
  ➞ tile_interleave_mpi -tx 128 -ty 2 -tz
  ➞ 16 -px 8 -py 4 -pz 1
```

Sidenote: Process pinning with ‘allsocks’ needs a longer time to start when compared with ‘allcores’, but on program execution these seem to perform equivalent.

6.6.2.2 Hybrid Intelmpi Process Pinning

The intelmpi implementation uses the ‘I_MPI_PIN_DOMAIN’ environment variable for hybrid implementations. The environment variable mentioned in section [6.6.2.1] should be ignored. Processor resources are now split into domains, where one process is mapped to a domain. Domains can be split by predefined granularities, such as socket or core or these can be defined explicitly. ‘I_MPI_PIN_ORDER’ defines in what order processes are divided among the domains. A good description is found in [16]. See execution configuration that yielded best performance for hybrid implementations:

```
# Mixed MPI execution on Cray
#
export OMP_NUM_THREADS=4

I_MPI_PIN_CELL=core I_MPI_PIN_DOMAIN=auto :
  ➞ compact mpiexec -n 8 bin/
  ➞ tile_interleave_mpi -tx 128 -ty 2 -tz
  ➞ 16 -px 8 -py 1 -pz 1
```
The *auto* value creates \( \frac{\text{cores}}{\text{processes}} \) domains within each socket on the node. This way all the cores are always inside an active domain. The complementing *compact* value defines that domain members should be close, which hopefully pins OMP threads to the same socket. `I_MPI_PIN_CELL` must be set to ‘core’.

### 6.6.2.3 Process Pinning on Cray XC40

Below are the different configurations used to start the runtime environment on the Cray. Note that ‘-n *num_procs*’ and ‘-N *num_tasks*’ are always used and defines the number of processes to be created in total and the number of tasks per node. Number of nodes used are derived by dividing the two terms with each other.

For pure MPI runs the CPU binding according to recommended settings in aprun’s programmers manual is used. See Script 5 for an example.

```bash
# Pure MPI execution on Cray
#
# This uses 2 compute nodes
aprun −cc cpu −n 64 −N 32 bin/
   ⇔ tile_interleave_mpi −tx 128 −ty 2 −tz
   ⇔ 16 −px 8 −py 8 −pz 1
```

*Script 5: Pure solver execution suggestion on Cray.*

For a hybrid run with one process per node the CPU binding is turned off. This is done because every compute node will have 1 process anyway (defined by the number of tasks ‘-N’). The OpenMP threads are pinned to each respective node according to the value set in ‘KMP_AFFINITY’. ‘-d’ simply defines the number of worker threads per process that are created, predefined in ‘OMP_NUM_THREADS’. See Script 6 for an example.

For a hybrid run with more then one process the *numa_node* value is used for CPU binding, which binds processes to compute nodes. The ‘-S’ flag is used to declare the number of processes per socket or as defined in the manual ‘per NUMA node’. The other values are the same as mentioned before. See Script 7 for an example.
# Hybrid MPI execution on Cray with 1 process

`export OMP_NUM_THREADS=32`
`export KMP_AFFINITY=scatter`

# This uses 32 nodes
`aprun -cc none -n 32 -N 1 -d 32 bin/`
`  tile_interleave_openmp_mpi -tx 128 -`
`  ty 2 -tz 16 -px 8 -py 4 -pz 1`

Script 6: Solver execution suggestion on Cray with 1 process per node.

# Hybrid MPI execution on Cray with more than 1 process

`export OMP_NUM_THREADS=4`
`export KMP_AFFINITY=none`
`aprun -cc numa_node -n 32 -N 4 -d 4 -S 4`
`  bin/tile_interleave_openmp_mpi -tx`
`  128 -ty 2 -tz 16 -px 8 -py 1 -pz 1`

Script 7: Hybrid solver execution suggestion on Cray.
6.6.3 Thread Count per Core

As discussed in section 2.2, we aim to execute one thread per core on the haswell processors and at least two per core on the Xeon Phi. For this implementation it appears that three or four threads yield the best performance when running on the Xeon Phi. Four is the best when running on one core and is considered in the next section, section 6.7.

6.7 Reference Times

All the parallel implementations are compared to the same reference times on their respective platform. The time always considered is the actual time taken for the solving of the vector field to the ending timestep. This means that it is not program execution, yet actual computing time that is considered.

Normally a sequential implementation is used without any OpenMP overhead. However, the sequential implementation performed worse then a single threaded OpenMP version. Using Intels VTune Amplifier 2015 and monitoring with a “General Exploration” profile, it became clear that the process running the sequential implementation was migrating between the processor sockets, implying remote main memory accesses. The “remote DRAM” access was alarmingly high in the VTune program. The single threaded OpenMP implementation only had half the time in comparison. Most likely this is because OpenMP pins the process to a socket or core and there was only local memory accesses made.

On the Xeon Phi the OpenMP implementation executed on 1 core with 4 threads.
<table>
<thead>
<tr>
<th>Platform</th>
<th>Time (s)</th>
<th>Config</th>
</tr>
</thead>
</table>
| Host         | 52.98    | KMP\_AFFINITY=scatter
OMP\_NUM\_THREADS=1
tile\_openmp\_solver -tx 128-
ty 2 -tz 16                        |
| Cray Node    | 59.75    | KMP\_AFFINITY=scatter
OMP\_NUM\_THREADS=1
tile\_openmp\_solver -tx 128-
ty 2 -tz 16                        |
| Xeon Phi     | 265.916  | KMP\_AFFINITY=compact
KMP\_PLACE\_THREADS=1c,4t
OMP\_NUM\_THREADS=4
mic\_tile\_openmp\_solver -tx 1024 -ty 1 -tz 1                       |

Table 1: Reference times for speedup calculations

6.8 Speedup and Efficiency Graphs

Speedup graphs are created from relating the time of different runs to a reference time, in our case they are defined in section 6.7. The speedup defines a relative measure on how much faster a run was compared to the reference. The times compared are always of the same platform, meaning that no speedup calculations will be made with Xeon Phi compared to the host reference time or vice versa.

Each implementation defined in section 6.3 will be compared to its platform specific reference time. This results in one speedup graph per platform. In each speedup graph there will be a theoretical speedup line, labelled as ‘theoretical’.

An efficiency graph is also created to illustrate how the implementations perform in terms of expected speedup. Each efficiency graph contains a line labelled as ‘100\%’ where ‘100\%’ represents perfect efficiency.

The roofline model presented in section 6.9 will be used to discuss the bottlenecks of the implementation in the discussion and related to potential issues seen in the results.

6.9 Roofline Model

According to [5] memory bandwidth is for the next foreseeable future a typical bottleneck when it comes to implementa-
tions with low FLOP to bytes loaded ratio. The roofline performance model from \cite{5} relates operational intensity with performance for a given platforms memory bandwidth and peak performance \cite{11}.

Operational intensity is the amount of work done divided by the memory traffic of a program \cite{11}. This is the same for all optimized implementations as the compute kernel is always the same, if it would be changed then the work has to be reevaluated. Performance is the raw number of FLOPs that the implementation can do per second. Operational intensity will be graphed against the performance, where different memory bandwidths of L2-cache, LLC or main memory determine the maximum possible performance. Due to limited time, the roofline model will not be used to model the Cray system.

The possible performance roof for a certain operational intensity is defined as:

\[
P_{\text{roof}} = \min(P_{\text{peak}}, I \times B)
\] (31)

Where \(I\) is the operational intensity and \(B\) is the memory bandwidth of the target memory. \(B\) must be measured. \(I\) is defined as:

\[
I = \frac{W}{Q} = \frac{\text{FLOP}}{\text{datatraffic}}.
\] (32)

Where \(W\) is the work, which is the number of FLOPs done in the compute kernel, and \(Q\) is the memory traffic done in the compute kernel.

For this performance evaluation we estimate all the values by looking at the code. We define the operational intensity of the implementation using the values derived earlier in eq. (25) and eq. (27) in section 5.3 which equates to:

\[
I = \frac{378}{1368 + 24} = 0.272 \text{ flop/byte}
\] (33)

The value of eq. (33) already suggests that there is a lot of memory bandwidth consumed per FLOP. The peak performance of a computational node can be summarized as:

\[
P_{\text{peak}} = \text{flop throughput} \times \text{frequency} \times \text{cores} \times \text{sockets}
\] (34)

For the Xeon Phi we can get the value from section 2.2. We remember that the Xeon Phi can do 8 Fused-Multiply-Add in-
struction (FMA) double-precision operations, which are 16 FLOP. The peak thus totals to:

\[ P_{\text{phi-peak}} = (2 \times 8) \times 1.238 \times 61 \times 1 = 1208 \text{GFLOP/s} \]  

Section 2.2 also discussed that two FMA instructions can be executed within one cycle for the Haswell processor. Therefore 16 double-precision FLOP are possible in one cycle. The peak on the host node thus totals to:

\[ P_{\text{host-peak}} = (4 \times 4) \times 2.3 \times 18 \times 2 = 1325 \text{GFLOP/s} \]  

Now in order to compare the operational intensity with the performance yielded we create a performance roof using eq. (31). The performance roofline can suggest a memory bound or a core bound implementation, for a specific target memory (that determines the roof). However first the memory bandwidth must be measured.

6.9.1 STREAM benchmark

Memory bandwidth measurements were done using the STREAM benchmark, provided by NERSC. The compilation for the Xeon Phi were done according to [32]. In order to measure L2 or L3 performance the array size was altered by defining the macro value \( N \) on compilation. The triad compute kernel was considered, as it is the most equivalent to our stencil computations.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Target Memory</th>
<th>Array Size</th>
<th>Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xeon Phi</td>
<td>L2 cache</td>
<td>1200000</td>
<td>369.4</td>
</tr>
<tr>
<td>Xeon Phi</td>
<td>RAM</td>
<td>8000000</td>
<td>176.4</td>
</tr>
<tr>
<td>Haswell</td>
<td>L2 cache</td>
<td>280000</td>
<td>671.1</td>
</tr>
<tr>
<td>Haswell</td>
<td>L3 cache</td>
<td>2500000</td>
<td>444.6</td>
</tr>
<tr>
<td>Haswell</td>
<td>RAM</td>
<td>8000000</td>
<td>69.1</td>
</tr>
</tbody>
</table>

Table 2: Memory bandwidths for Xeon Phi and Xeon E5-2699 v3

Using the expression eq. (31) we can now draw a roofline for every memory bandwidth on a graph comparing operational intensity to performance. The roofline graphs have a logarithmic scale in both X- and Y-axis to make it visually more pleasing. In the next section the roofline graphs are presented together with the speedup and efficiency graphs.
RESULTS

This chapter summarizes the results after the benchmarks were done. Section 7.1 presents the roofline model and points out some observations. Section 7.2 presents the best results from a OpenMP, pure MPI and a hybrid MPI implementation executed on the host. Section 7.3 presents the results of running OpenMP on a Xeon Phi card.

7.1 ROOFLINE MODELS

Table 3 presents the GFLOP/s used in the roofline graph in Result 1 for the Xeon Phi. An optimized version is compared to a version not using loop tiling and auto-vectorization turned on, labelled as solver. no_vec_tile_solver represents a tiling implementation without auto-vectorization. no_vec_solver has no loop tiling and no auto vectorization. Streaming stores is active and prefetching is done by the compiler if possible. optimized is the best performing implementation. All runs are using OpenMP and execute with 4 threads on each core, with 60 cores in total.

NOTE: no auto vectorization still generated vector instructions in the assembler code. This is an interesting behaviour and could be investigating in future work.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>GFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimized</td>
<td>92.0</td>
</tr>
<tr>
<td>solver</td>
<td>47.6</td>
</tr>
<tr>
<td>no_vec_tile_solver</td>
<td>18.5</td>
</tr>
<tr>
<td>no_vec_solver</td>
<td>8.0</td>
</tr>
</tbody>
</table>

Table 3: Performance used in Xeon Phi roofline model.
Table 4 presents the GFLOP/s used in the roofline graph in Result 2 for the Haswell. The naming scheme is similar as just presented in the beginning of this section. All the runs are OpenMP based that execute with 36 threads scattered on all cores of the host system.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>GFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimized</td>
<td>292.0</td>
</tr>
<tr>
<td>no_vec_tile_solver</td>
<td>115.4</td>
</tr>
<tr>
<td>solver</td>
<td>111.2</td>
</tr>
<tr>
<td>no_vec_solver</td>
<td>106.4</td>
</tr>
</tbody>
</table>

Table 4: Performance used in Haswell roofline model.
Result 2: Roofline model for Haswell processor, Xeon E5-2699 v3

7.2 HASWELL

Both graphs Results [3] and [4] present three different implementations, the OpenMP, pure MPI and a mixed implementation. OpenMP has for more than 16 cores the best efficiency and also experiences the highest speedup, up to 29 (if comparing with 32 cores used). The pure MPI implementation’s efficiency degrades the fastest and has in general the worst speedup.

Table Table 5 summarizes the best performance yielded when running on 32 cores.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>GFLOP/s</th>
<th>Speedup</th>
<th>Speedup Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP (36t)</td>
<td>292.0</td>
<td>30.4</td>
<td>84.5%</td>
</tr>
<tr>
<td>OpenMP (32t)</td>
<td>277.0</td>
<td>28.9</td>
<td>90.2%</td>
</tr>
<tr>
<td>Mixed (2p,16t)</td>
<td>266.8</td>
<td>27.8</td>
<td>86.9%</td>
</tr>
<tr>
<td>MPI (32p)</td>
<td>233.8</td>
<td>24.4</td>
<td>76.1%</td>
</tr>
</tbody>
</table>

Table 5: Best performance for each implementation for host system.
Result 3: Speedup graph of host machine with two Xeon E5-2699 v3 processors. Reference values generated with 'tile_openmp_solver' using tiling configurations 128x2x16.

Result 4: Efficiency graph of host machine with two Xeon E5-2699 v3 processors. Reference values generated with 'tile_openmp_solver' using tiling configurations 128x2x16.
Both graphs in Results 5 and 6 show the execution of the OpenMP implementation with four different configurations. Each execution utilizes 60 cores, yet has either 1, 2, 3 or 4 threads running per core. Executing with 4 threads yields the best performance, yet for large number of cores executing with 3 threads per core is almost as efficient.

Table 6 summarizes the speedup, performance and efficiency relative to a single core running with 4 threads.

<table>
<thead>
<tr>
<th>CoreThreads</th>
<th>Cores</th>
<th>Speedup</th>
<th>GFLOP/s</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0.51</td>
<td>0.97</td>
<td>50.7%</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.81</td>
<td>1.54</td>
<td>80.5%</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0.96</td>
<td>1.82</td>
<td>95.6%</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1.00</td>
<td>1.91</td>
<td>100%</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>4.9</td>
<td>9.3</td>
<td>48.9%</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>7.9</td>
<td>15.0</td>
<td>78.5%</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>8.9</td>
<td>16.9</td>
<td>88.5%</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>9.1</td>
<td>17.4</td>
<td>91.2%</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>9.8</td>
<td>18.6</td>
<td>48.8%</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>15.6</td>
<td>29.7</td>
<td>77.9%</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>17.3</td>
<td>33.1</td>
<td>86.6%</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>17.7</td>
<td>33.8</td>
<td>88.7%</td>
</tr>
<tr>
<td>1</td>
<td>40</td>
<td>19.6</td>
<td>37.5</td>
<td>49.1%</td>
</tr>
<tr>
<td>2</td>
<td>40</td>
<td>31.0</td>
<td>59.1</td>
<td>77.5%</td>
</tr>
<tr>
<td>3</td>
<td>40</td>
<td>34.4</td>
<td>64.9</td>
<td>85.1%</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>34.6</td>
<td>66.1</td>
<td>86.6%</td>
</tr>
<tr>
<td>1</td>
<td>60</td>
<td>29.6</td>
<td>56.4</td>
<td>49.3%</td>
</tr>
<tr>
<td>2</td>
<td>60</td>
<td>46.5</td>
<td>88.7</td>
<td>77.5%</td>
</tr>
<tr>
<td>3</td>
<td>60</td>
<td>50.1</td>
<td>95.6</td>
<td>83.5%</td>
</tr>
<tr>
<td>4</td>
<td>60</td>
<td>50.3</td>
<td>96.0</td>
<td>83.9%</td>
</tr>
</tbody>
</table>

Table 6: Results from Xeon Phi experiments.
Result 5: Speedup graph of one Xeon Phi card. Reference values generated with 'mic_prefetch_tile_openmp_solver' using tiling configurations 1024x1x1.

Result 6: Efficiency graph of one Xeon Phi card. Reference values generated with 'mic_prefetch_tile_openmp_solver' using tiling configurations 1024x1x1.
This section presents results from the execution of a pure MPI and hybrid implementation on the Beskow cluster.

Result 7 presents the speedup of up to 512 cores (16 nodes) where the pure MPI solution performs the best at a speedup up to almost 170 with a 33.3% efficiency. The hybrid implementation with 16 processes and 2 threads per node lands at a speedup of about 132 with an efficiency of just below 30%. Efficiencies can be interpreted from Result 8.

Result 7: Speedup graph of Cray XC40 cluster. Reference values generated with 'tile_openmp_solver' (single threaded) using tiling configurations 128x2x16.
Result 8: Efficiency graph of Cray XC40 cluster. Reference values generated with 'tile_openmp_solver' (single threaded) using tiling configurations 128x2x16.

Result 9 presents the speedup of up to 4096 cores (128 nodes) where the pure MPI solution performs the best at a speedup up to almost 290 with a 56% efficiency. The hybrid implementation with 16 processes and 2 threads per node lands at a speedup of about 832 with an efficiency of just above 20%.

Result 9: Speedup graph of Cray XC40 cluster on 4x larger problem-set. Reference values generated with 'tile_openmp_solver' (single threaded) using tiling configurations 128x2x16.
Result 10: Efficiency graph of Cray XC40 cluster on 4x larger problem-set. Reference values generated with 'tile_openmp_solver' (single threaded) using tiling configurations $128\times2\times16$.

Result 11: Bar chart describing average performance per node on a Cray XC40 system. Different number of nodes are used and compared to Xeon Phi and the Haswell raw performance.
DISCUSSION

This chapter is a discussion about the observation made from the results presented in chapter 7. First we discuss the memory wall on the host system and the Xeon Phi. The second section is about how the two processors compare against each other. The last section discusses how the implementation performed in a cluster environment.

8.1 roofline model: the memory wall

The roofline model can illuminate possible bottlenecks of High Performance Computing (HPC) programs. Either the processor cannot process data fetched from memory fast enough and is limited by the peak performance of its processing capabilities or data cannot be fed in time to the processor for computations. Execution bottlenecks usually happens with high operational intensity, when the FLOP count is very large compared to data traffic. In our case there is low operational intensity, which we see in our roofline graphs Results 1 and 2 that may hint towards that the implementation will be bottlenecked by memory bandwidth.

8.1.1 Xeon Phi

The Xeon Phi roofline graph illustrates that the no_vec_solver and no_vec_tile_solver implementations (red square and yellow rhombus shape in Result 1) are under the RAM roofline, suggesting that these implementations are execution bound. No auto-vectorization is used that explains execution boundness, which is one of the Xeon Phi’s greatest assets for performance. The solver implementation (triangle shape in Result 1) utilizes auto-vectorization, yet no loop tiling, and is just below the roofline of RAM. This suggests that the RAM bandwidth of the system is the performance bottleneck. The small problem-set (see
section 6.2 is larger than all the L2-caches combined and thus it appears that the L2-cache suffers from cache thrashing, discussed in section 5.4.5. The *optimized* implementation utilizes loop tiling that leads to L2 cache bandwidth being a bottleneck for performance instead, suggesting that the tiling strategy is vital for performance as is vectorization.

On a sidenote: Explicit prefetching gives marginally more performance than using the compiler heuristics to create prefetch instructions, however the difference is minimal so it was not executed separately and instead enabled in all experiments. As was streaming stores.

8.1.2 Haswell

The Haswell roofline graph illustrates that the *no_vec_solver, solver* and *no_vec_tile_solver* (square, triangle and rhombus shapes in Result 2) implementations performance is bound by L3 cache bandwidth. As suggested in Table 4 the *no_vec_solver* has a small edge against the other two implementations. It might be that the problem-set plays nicely with the L3 cache size, allowing a good hit/miss ratio and good cooperation between threads fetching overlapping data. Its apparent that loop tiling for the latter three implementations is not relevant suggesting that these are execution bound as there is no vectorization done. The *optimized* implementation has both autovectorization and loop tiling and is therefore able to go pass the roofline of the L3 cache due to increased processing capabilities. It also surpasses the L2 cache hinting towards a L1 memory bound situation, but this requires another experiment to investigate the L1 cache on the Haswell.

Estimated data traffic values may differ to more concrete values that can be measured, which influences the value of operational intensity and thus the possible performance with respect to memory. The memory bandwidths from the STREAM benchmark may very well also suggest too pessimistic results, lowering the rooflines.

8.2 Xeon Phi vs Haswell

The implementation in its best state suggests that it performs fine on both processors. As seen in Result 3 the best performance was reached with OpenMP, which makes the results on both processors highly comparable, as the best implemen-
tations are codewise almost identical. Next follows a discussion around speedup and efficiencies, load balancing and task granularity.

The Haswell OpenMP implementation exceed the MPI and hybrid implementation in performance. In fact, this performance difference was reached when utilizing smaller task granularity motivated in section 5.5.1.2. Thread pinning method used is scattered that will put every second thread on a different socket. Section 5.5.1.2 suggested that a fine task granularity will have a collaborative benefit in terms overlapping cachelines in L3, which is partially lost when pinning the threads using scatter. The implementation might benefit from a compact scheme, yet with a core granularity approach instead, which could maximize the amount of cachelines reused in other threads via L3 access. Having that said, the roofline model suggests that the implementation is not bound by L2 cache bandwidth, which makes us believe that this issue is not so significant.

The MPI implementations feature interleaved communication with computations. As the hardware supports asynchronous I/O, the data can actually be sent concurrently with computations. The addition of interleaving added a boost in all variants: pure MPI and hybrid. The code that computes the overlapping area, after boundaries have been transferred to the ghost cells, adds an extra pass over the mesh, which partially can cause degraded performance if vectorized as the data blocks are too short. In other words, the MPI implementation might have to be revised in order to compete with the OpenMP implementation. A hybrid approach does however come really close to the OpenMP implementation in a setup of two processes and 16 threads, having almost the same efficiency and speedup as a 32 threaded Openmp run.

The performance of the Xeon Phi is the best when utilizing three or four threads per core, which we can see in Table 6. The GFLOP/s values are most of the time favouring four threads per core. One thread per core behaves as discussed in section 2.2, where we mention that using only one thread per core halves the performance. What's very impressive is the scaling in terms of efficiency of the Xeon Phi, when compared to the Haswell. The Xeon Phi runs with almost twice as many cores and manages to have the same efficiency.

The Haswell implementation reached the highest performance compared to the Xeon Phi, with almost three times as many GFLOP/s than the Xeon Phi results. Apart from the raw per-
formance results, both processors manage to end up with the same efficiency at around 85% (see Tables 5 and 6). The Haswell has a couple of advantages for this implementation speaking for its higher performance. One of them being that it has a higher clock frequency, allowing it to issue more instructions per seconds. On top of that the Haswell features out-of-order execution and can interleave computational instructions with load instructions, where the Xeon Phi instead stalls. Lastly we saw in the roofline models that the Haswell has a much higher bandwidth in all its cache levels. The only advantage going for the Xeon Phi is its higher bandwidth RAM access. Considering that the Xeon Phi Knights Corner is two years older than the Haswell architecture it simply seems that the technological advances of the Haswell puts the Xeon Phi in its shadows.

The computation of a stencil in Code 13 are for the moment divided up into three expressions that are very long and in that sense complex. As the Xeon Phi is dependent on the instruction order created by the compiler, it could improve performance if we would try to simply the expressions written. One approach would be to write all FLOPs made as single expressions, possibly allowing the compiler to optimize the order. Furthermore the usage of compact stencil, instead of a classical stencil, might add more cache utilization that could be beneficial for the Xeon Phi and possibly the Haswell as well. This however changes the implementation.

8.3 Scaling on Cray XC40 System

In Result 9 for the problem-set big we see that the MPI implementations experience linear speedup that is far from optimal. For the small problem-set in Result 7 it appears that the problem-set small results in to little work for a larger number of nodes, causing MPI communication to be a larger part of program execution (reducing performance). Comparing the small problem-set with the big problem-set we see signs of weak scaling according to Gustafsson’s law as parallel speedup becomes more efficient for a larger problem-set. The pure MPI implementation drops to almost 30% efficiency when using 512 cores, equivalent to 16 computing nodes on the Cray XC40. When using problem-set big we see that efficiency is at almost 50% with 4096 cores, equivalent to 128 compute nodes.

Result 11 illustrates the average node performance for 2, 4, 8, 16 and 32 Cray nodes compared to the host system and the
Xeon Phi. We know that the MPI implementation becomes less efficient as more nodes are added on, so we expect a lower average performance compared to the host system. Problem-set small is too small as suggested in the last paragraph and experiences MPI communications overhead. From 8 nodes and upwards there is a clear performance drop that could relate to suboptimal MPI communication, as the implementation has not been optimized from this perspective.

The only potential optimization steps that this implementation does is introducing a virtual topology to make it possible for the runtime system to pin processes to nodes with respect to the topology. As mentioned in section 5.5.2.2 this is most likely not the case. We see performance potential for the pure MPI implementation, yet another project would have to be made that experiments on different techniques to ensure that close communication partners are indeed neighbours in the virtual topology to ensure optimal data transfers. Its quite possible that suboptimal process placements cause the bad performance for the hybrid implementation too, but this must be investigated in future work.

8.4 conclusion

When running on a single Haswell compute node the implementation does perform well. With few optimizations the Haswell becomes L3 bound suggesting that good memory layout and data access set a solid performance foundation. Using loop tiling and vectorization performance goes beyond any measured rooflines. The Xeon Phi roofline illustrates the importance of cache utilization, but mostly highlights that vectorization is very important for performance. We want to emphasize that the in-order execution of Xeon Phi and its lower clock frequency are factors to consider when comparing the Xeon Phi with the Haswell. The next generation of the MIC will be the Xeon Phi Knights Landing that will be an out-of-order stand-alone processor [34] and thus some of the factors we mention disappear. Considering that the implementation performed well on the Haswell and scaled well using OpenMP on the Xeon Phi, it might indeed lead to better results on the next generation of MIC.

We see signs of weak scaling for the pure MPI implementation on the Cray XC40. However, further work is needed to optimize the communication between MPI processes to opti-
mize the performance. Most likely further investigation is also required to find out why the hybrid implementation performs so much worse than the pure MPI implementation.

A solver has been implemented with several optimization strategies that runs efficiently on a Haswell compute node and that also performs well on the Xeon Phi, if we consider the limitations of the Xeon Phi discussed.
FUTURE WORK

Further work needs to be done to possibly optimize performance on the Xeon Phi. Simplification of the stencil computations was suggested and also other performance tuning can most likely be applied that wasn’t tested in this project. MPI execution on several Xeon Phi cards or in an MPI run-time environment with different processors, such as the host processor and a couple of Xeon Phi cards would be an interesting combination to see.

We saw in chapter 4, related work, that the Xeon Phi has been used successfully in linear algebra contexts, which seems promising for implicit FDM solvers. This would also be a nice comparison to this project.

The MPI implementation also requires more attention and must be optimized for running on a cluster. A first attempt would most likely be to implement a function that can be use to assign nearest neighbours to a specific process in order to improve MPI communication.

When the Xeon Phi Knights Landing main processor is released it will be interesting to run the same experiments as done in this project to compare what the new architecture offers.
BIBLIOGRAPHY


## APPENDIX

### A.1 RESULTS HASWELL

<table>
<thead>
<tr>
<th>threads</th>
<th>solver</th>
<th>time</th>
<th>speedup</th>
<th>GFLOP/s</th>
<th>efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>tile_solver_openmp</td>
<td>52.8g</td>
<td>1</td>
<td>9.53</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>tile_solver_openmp</td>
<td>27.16</td>
<td>1.95</td>
<td>18.68</td>
<td>97.37%</td>
</tr>
<tr>
<td>4</td>
<td>tile_solver_openmp</td>
<td>13.70</td>
<td>3.86</td>
<td>37.06</td>
<td>96.58%</td>
</tr>
<tr>
<td>8</td>
<td>tile_solver_openmp</td>
<td>6.84</td>
<td>7.73</td>
<td>74.15</td>
<td>96.62%</td>
</tr>
<tr>
<td>16</td>
<td>tile_solver_openmp</td>
<td>3.45</td>
<td>15.31</td>
<td>146.85</td>
<td>95.67%</td>
</tr>
<tr>
<td>32</td>
<td>tile_solver_openmp</td>
<td>1.83</td>
<td>28.88</td>
<td>277.01</td>
<td>90.24%</td>
</tr>
<tr>
<td>36</td>
<td>tile_solver_openmp</td>
<td>1.74</td>
<td>30.43</td>
<td>291.87</td>
<td>84.51%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>threads</th>
<th>solver</th>
<th>time</th>
<th>speedup</th>
<th>GFLOP/s</th>
<th>efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>tile_interleave_mpi</td>
<td>52.8g</td>
<td>1</td>
<td>9.59</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>tile_interleave_mpi</td>
<td>30.24</td>
<td>1.75</td>
<td>16.78</td>
<td>87.44%</td>
</tr>
<tr>
<td>4</td>
<td>tile_interleave_mpi</td>
<td>14.35</td>
<td>3.69</td>
<td>35.35</td>
<td>92.14%</td>
</tr>
<tr>
<td>8</td>
<td>tile_interleave_mpi</td>
<td>6.86</td>
<td>7.71</td>
<td>73.96</td>
<td>96.37%</td>
</tr>
<tr>
<td>16</td>
<td>tile_interleave_mpi</td>
<td>3.66</td>
<td>14.45</td>
<td>138.62</td>
<td>90.31%</td>
</tr>
<tr>
<td>32</td>
<td>tile_interleave_mpi</td>
<td>2.17</td>
<td>24.37</td>
<td>233.80</td>
<td>76.16%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>threads</th>
<th>solver</th>
<th>time</th>
<th>speedup</th>
<th>GFLOP/s</th>
<th>efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq</td>
<td>tile_openmp_interleave_mpi</td>
<td>52.8g</td>
<td>1</td>
<td>9.59</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>tile_openmp_interleave_mpi</td>
<td>30.24</td>
<td>1.75</td>
<td>16.78</td>
<td>87.44%</td>
</tr>
<tr>
<td>4</td>
<td>tile_openmp_interleave_mpi</td>
<td>13.37</td>
<td>3.96</td>
<td>37.96</td>
<td>98.92%</td>
</tr>
<tr>
<td>8</td>
<td>tile_openmp_interleave_mpi</td>
<td>6.80</td>
<td>7.78</td>
<td>74.63</td>
<td>97.25%</td>
</tr>
<tr>
<td>16</td>
<td>tile_openmp_interleave_mpi</td>
<td>3.50</td>
<td>15.09</td>
<td>144.78</td>
<td>94.32%</td>
</tr>
<tr>
<td>32</td>
<td>tile_openmp_interleave_mpi</td>
<td>1.90</td>
<td>27.81</td>
<td>266.83</td>
<td>86.92%</td>
</tr>
</tbody>
</table>
This are images from a 8x8 simulation. X flow goes from left to right and starts with a flow magnitude above zero where its red. Y flow goes from bottom to top and also starts with a flow above zero where its red. Notice how the flow in both images tends to the top right.

**X-Flow**

\[ T = 0 \]

\[ T = 0.3 \]
$T = 0.6$

$T = 0.9$
$T = 2.0$

\[ \text{Y-Flow} \]

$T = 0$
$T = 0.3$

$T = 0.6$
$T = 0.9$

$T = 2.0$
LIST OF CODE SNIPPETS

1. Compiler must assume that pointers can alias the same memory. ........................................... 12
2. ivdep pragma ignores any data dependencies. .......................................................... 12
3. restrict keyword tells compiler that declared variables are not aliased. ........................................... 13
4. No flow dependency (due to restrict), yet output dependency instead. ........................................... 13
5. The array 'a' is originally aligned to 64 bytes, however the increment of 'val1' and access to index 'val2+1' may not be. ......................................................... 14
6. '_assume_aligned(..)' makes sure that the compiler understands that 'a[i]' is an aligned access. 'a[val2+i]' is cannot be determined as an aligned access. ......................................................... 15
7. '_assume(..)' makes sure that the compiler understands that 'val2' implies an aligned access on an aligned pointer. 'a[val2+0]' is an aligned access, so is 'a[0]', thus the loop can be vectorized with an aligned access. ......................................................... 16
8. Defining all vector stores in a loop as nontemporal. ......................................................... 17
9. Assume that loop is vectorized, then prefetching of a and b to L1 and L2 cache is hinted to the compiler, with vector distance 1 and 4 resp. b is not prefetched. ......................................................... 20
10. Instance defining the domain to solve. ......................................................... 43
11. Instance defining the distances between points in all three dimensions and the value of one timestep (dt). ......................................................... 43
12. Snippet of the outermost loop iterating over time. Each iteration yields a new timestep 'it' where \[ t = it \times dt \]. Each time iteration loops over all points and updates these accordingly. See Code 13 for more details regarding approx_u(..). ......................................................... 45
Snippet of the inline function that updates a point in mesh of x, y and z component. See Code 14 for more details regarding \texttt{approx\_sum\_d(\ldots)} and \texttt{approx\_sum\_d2(\ldots)}.

Snippet of macro definitions expanded in Code 13.

\texttt{STRIDE} is either the x-stride, y-stride or z-stride.

Code snippet illustrating how to provide loop specific compiler directives about prefetching.

Loop tiling with six nested loops.

Loop tiling with 4 nested loops.

Compiler hints place to allow auto-vectorization to vectorize appropriately.

Parallelization with one OpenMP parallel region.

Parallelization with one OpenMP parallel region and for-constructs.

\texttt{MPI\_Params} struct containing MPI related information needed in runtime by the solver.

Declaration, definition and commit of MPI Derived Datatypes.

Macro for boundary exchange. Start receive communication and then the send communications.

Macro for waiting for MPI communication to finish.

Function wrapper of MPI\_Isend(\ldots) that sends data, which read access pattern is defined by the supplied MPI\_Datatype.

Function wrapper of MPI\_Irecv(\ldots) that receives data, which write access pattern is defined by the supplied MPI\_Datatype.

MPI implementation. See Codes 26 and 27 for expansion of macros.
## LIST OF SCRIPTS

<table>
<thead>
<tr>
<th></th>
<th>Either always use streaming stores, let the compiler decide when to use them or turn streaming stores completely off.</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Execution examples for executing solver with different command-line inputs</td>
<td>41</td>
</tr>
<tr>
<td>3</td>
<td>Default configuration file.</td>
<td>42</td>
</tr>
<tr>
<td>4</td>
<td>Different ways to execute different configurations of implementation.</td>
<td>87</td>
</tr>
<tr>
<td>5</td>
<td>Pure solver execution suggestion on Cray.</td>
<td>90</td>
</tr>
<tr>
<td>6</td>
<td>Solver execution suggestion on Cray with 1 process per node.</td>
<td>91</td>
</tr>
<tr>
<td>7</td>
<td>Hybrid solver execution suggestion on Cray.</td>
<td>91</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Figure 1</td>
<td>Overview of the Xeon Phi. Cores are connected over the ring interconnect.</td>
<td>4</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Overview of a Haswell compute node with two processors. A processor has local RAM and multiple cores all share L3 cache. Two processors communicate over QPI.</td>
<td>5</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Simple illustration of the Xeon Phi execution pipeline.</td>
<td>6</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Simple illustration of the haswell execution pipeline.</td>
<td>7</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Higher level example: Normal scalar addition compared to SIMD addition.</td>
<td>8</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Higher level example: Normal scalar addition. It implies 8 loads for each element required in the scalar operation and 8 stores back to cache for each individual result.</td>
<td>9</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Higher level example: SIMD addition. Only 2 vector loads and one vector store compared to scalar addition in fig. 6.</td>
<td>9</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Higher level example: Non-continuous data makes data fetching for SIMD operations more complex. Data is in this case present in 8 different cachelines instead of 1. Modern processors support some kind of 'gather' instructions to handle this more efficiently.</td>
<td>10</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Higher level example: Non-continuous data makes data storing of a resultant vector more complex. Data should in this case be stored in 8 different cachelines instead of 1. Modern processors support some kind of 'scatter' instructions to handle this more efficiently.</td>
<td>11</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Unaligned data access results in additional stores and loads.</td>
<td>14</td>
</tr>
<tr>
<td>Figure 11</td>
<td>A 3D mesh, where the cubes illustrate points.</td>
<td>25</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>A vector field, where each point has a certain flow.</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>7-point computational stencil for second order accuracy approximation.</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>3D mesh with 7-point computational stencil (shaded cells).</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>25-point computational stencil for eighth order accuracy approximation.</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>A 3D mesh with boundaries (light shadings) and a computational domain (darker shadings).</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Memory layout of a 3D mesh. Data is aligned to x-axis, then y-axis and lastly the z-axis (thus the single stride, y-stride and z-stride).</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Very simplified view of a part of L1 D-cache, data viewed are points from a computational 25-point stencil centered at (i,j,k). All the indices are relative to (i,j,k). Bold text and non-black elements are part of stencil.</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>Very simplified view of a part of L1 D-cache, data viewed are points from a computational 25-point stencil centered at (i+1,j,k). The data is already in cache from a previous computation. All the indices are relative to (i,j,k). Bold text and non-black elements are part of stencil.</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Computations of two neighbouring stencils overlapping, 2D example.</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>&quot;Life cycle&quot; of a cacheline that holds neighbouring points in x-axis. Numbers on each line correspond to a relative position to index i, when j and k are fixed. It will yield hits for up to 16 different computational stencils. This life cycle corresponds to the life cycle of the blue boxes in fig. 18 and fig. 19.</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Static scheduling with default granularity. If unlucky there is no working-set overlapping and mutual cachelines between threads are not present.</td>
<td></td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Static scheduling, yet with fine granularity (4 in this case). If lucky threads will have overlapping working-sets and experience more cache benefits in L3.</td>
<td>65</td>
</tr>
<tr>
<td>24</td>
<td>Domain split into four subdomains. Each subdomain is bound to a process.</td>
<td>67</td>
</tr>
<tr>
<td>25</td>
<td>Boundary exchange at X-Y 2D slice. Overlapping area is sent to a receiving part that copies it to its ghost cell area.</td>
<td>68</td>
</tr>
<tr>
<td>26</td>
<td>Boundary exchange between processes in 3D. Overlapping area is sent to a receiving part, which copies it to its ghost cell area.</td>
<td>68</td>
</tr>
<tr>
<td>27</td>
<td>A one-dimensional virtual MPI Cart topology. Processes are structured as a cartesian structure. Arrows indicate communication partners.</td>
<td>69</td>
</tr>
<tr>
<td>28</td>
<td>A 2D virtual MPI Cart topology. Processes are structured as a cartesian structure. Arrows indicate communication partners.</td>
<td>70</td>
</tr>
<tr>
<td>29</td>
<td>MPI communication of non-continuous data without MPI Derived Datatype.</td>
<td>73</td>
</tr>
<tr>
<td>30</td>
<td>MPI communication of non-continuous data using MPI Derived Datatype.</td>
<td>73</td>
</tr>
<tr>
<td>31</td>
<td>Innermost cells can be computed before MPI communication finishes. Overlapping area can first be computed when boundary exchange is done.</td>
<td>79</td>
</tr>
</tbody>
</table>
**LIST OF RESULT GRAPHS**

| Table 1 | Reference times for speedup calculations  | 93 |
| Table 2 | Memory bandwidths for Xeon Phi and Xeon E5-2699 v3 | 95 |
| Table 3 | Performance used in Xeon Phi roofline model. | 97 |
|        | 1 Roofline model for native Xeon Phi execution. | 98 |
| Table 4 | Performance used in Haswell roofline model. | 98 |
|        | 2 Roofline model for Haswell processor, Xeon E5-2699 v3 | 99 |
| Table 5 | Best performance for each implementation for host system. | 99 |
|        | 3 Speedup graph of host machine with two Xeon E5-2699 v3 processors. Reference values generated with 'tile_openmp_solver' using tiling configurations 128x2x16. | 100 |
|        | 4 Efficiency graph of host machine with two Xeon E5-2699 v3 processors. Reference values generated with 'tile_openmp_solver' using tiling configurations 128x2x16. | 100 |
| Table 6 | Results from Xeon Phi experiments. | 101 |
|        | 5 Speedup graph of one Xeon Phi card. Reference values generated with 'mic_prefetch_tile_openmp_solver' using tiling configurations 1024x1x1. | 102 |
|        | 6 Efficiency graph of one Xeon Phi card. Reference values generated with 'mic_prefetch_tile_openmp_solver' using tiling configurations 1024x1x1. | 102 |
|        | 7 Speedup graph of Cray XC40 cluster. Reference values generated with 'tile_openmp_solver' (single threaded) using tiling configurations 128x2x16. | 103 |
|        | 8 Efficiency graph of Cray XC40 cluster. Reference values generated with 'tile_openmp_solver' (single threaded) using tiling configurations 128x2x16. | 104 |
|        | 9 Speedup graph of Cray XC40 cluster on 4x larger problem-set. Reference values generated with 'tile_openmp_solver' (single threaded) using tiling configurations 128x2x16. | 104 |
Efficiency graph of Cray XC40 cluster on 4x larger problem-set. Reference values generated with `tile_openmp_solver` (single threaded) using tiling configurations 128x2x16.

Bar chart describing average performance per node on a Cray XC40 system. Different number of nodes are used and compared to Xeon Phi and the Haswell raw performance.
## ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface.</td>
<td>1</td>
</tr>
<tr>
<td>DCU</td>
<td>Data Cache Unit Prefetcher.</td>
<td>19</td>
</tr>
<tr>
<td>FDM</td>
<td>Finite Difference Method.</td>
<td>24, 26, 28, 38, 113</td>
</tr>
<tr>
<td>FLOP</td>
<td>Floating-point Operations.</td>
<td>45, 47, 94, 95, 107, 110</td>
</tr>
<tr>
<td>FLOPs</td>
<td>Floating-point Operations Per Second.</td>
<td>47</td>
</tr>
<tr>
<td>FMA</td>
<td>Fused-Multiply-Add instruction.</td>
<td>7, 11, 37, 94</td>
</tr>
<tr>
<td>FTCS</td>
<td>Forward in Time Central in Space.</td>
<td>29, 34, 39</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose computing on Graphical Processing Units.</td>
<td>37</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphical Processing Units.</td>
<td>37, 38</td>
</tr>
<tr>
<td>HPC</td>
<td>High Performance Computing.</td>
<td>37, 107</td>
</tr>
<tr>
<td>IPO</td>
<td>Interprocedural Optimization.</td>
<td>18</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture.</td>
<td>3</td>
</tr>
<tr>
<td>LLC</td>
<td>Last Level Cache.</td>
<td>19</td>
</tr>
<tr>
<td>LRU</td>
<td>Least Recently Used.</td>
<td>18</td>
</tr>
<tr>
<td>MIC</td>
<td>Many-Integrated-Core.</td>
<td>1, 4, 111</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface.</td>
<td>1, 2, 37, 61, 66, 79, 81</td>
</tr>
<tr>
<td>NERSC</td>
<td>National Energy Research Scientific Computing Center.</td>
<td>95</td>
</tr>
<tr>
<td>NUMA</td>
<td>Non-Uniform Memory Access.</td>
<td>44, 64, 87, 90</td>
</tr>
<tr>
<td>OpenMP</td>
<td>OpenMP.</td>
<td>37, 61, 62, 79</td>
</tr>
<tr>
<td>PDE</td>
<td>Partial Differential Equation.</td>
<td>23</td>
</tr>
<tr>
<td>RHS</td>
<td>Right Hand Side.</td>
<td>12</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Input Multiple Data.</td>
<td>8, 9, 11, 16, 37</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric Multi-Processing.</td>
<td>3, 37</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>TD</td>
<td>Tag Directory.</td>
<td></td>
</tr>
<tr>
<td>VPU</td>
<td>Vector Processing Unit.</td>
<td></td>
</tr>
<tr>
<td>x86</td>
<td>x86 Instruction-Set.</td>
<td></td>
</tr>
</tbody>
</table>
GLOS SARY

8-way-associative Cache associativity is the number of cachelines that can be inserted in a cache block.. 18

array of struct An array containing structs.. 50

boundary domain The domain of points that is not included in the computational domain, which is initially set and fixed by the dirichlet boundary condition.. 39 43 44 48

cache A cache usually consider a temporary storage for recently accessed resources, which is suppose to speedup accesses on slower memory locations.. 4 18 21 50 53 55 58 132

cache pollution Cache pollution occurs when a cache is filled with cachelines that will not be used before these are evicted (unnecessary loads into cache). 59

cache trashing When data access leads to the cache eviction of cachelines currently in use.. 51 55

cache-miss A cache-miss occurs when the processor tries to load a cacheline from its cache, yet the data is not present.. 21

cacheline A cacheline is the smallest database size that is used in cache I/O. Typically 64 bytes in size.. 10 11 14 18 21 50 51 55 56 131 132
compact stencil

A compact stencil uses only its adjacent neighbouring points in its computation. This means that there are only strict number of points possible per dimension. One dimension uses 3, two dimension uses 9 and three dimensions uses 27..

computational domain

The domain of points that is going to be approximated in each timestep, which is initially set by the initial condition.

coprocessor

A processor that is not a main processor. Coprocessor are connected to a host system and do computations offloaded by a host.

D-cache

D-cache is a cache for data.

data locality principle

The principle of locality is divided up into two categories: spatial and temporal locality. Spatial locality principle refers to that data close to other data currently being accessed will most likely be accessed ‘soon’. Temporal locality is when data recently accessed is accessed again.

data race condition

When two or more threads write to shared variables in an unsynchronized manner, possible causing undefined program behavior.

data-dependency

If calculations in a loop are dependent on previous calculations then there is a data dependency.

dirichlet boundary condition

Fixed values on the boundary domain.
discretize
The process of converting something infinite and unbounded into something finite and bounded... \(26\)

**discretized**
See discretize \(24\)

**discretized space**
A bounded space with countable number of points.. \(24\)

**FMA**
Fused-Multiply-Add instruction. \(7\)

**hardware prefetcher**
A prefetcher built-in to the hardware that will prefetch memory from RAM to LLC, or from a cache level to its higher level caches.. \(19\)

**I-cache**
I-cache is a cache for instructions.. \(18\)

**initial condition**
The start values for a computational domain. . \(39\)

**intrinsic**
Intrinsics functions are functions whose implementation are handled by a compiler.. \(1\)

**load balancing**
Load balancing is the process of making sure that two or more threads receive a workload that is similarly computationally intense.. \(64\)

**micro-operation**
Micro operations usually perform basic operations on data currently stored in registers.. \(6\)

**QPI**
The interconnect between two processors in a compute node.. \(4\)

\(141\)
replacement-policy  
A replacement policy in terms of cache is the policy determining what cachelines are going to be evicted when the cache is full and another cacheline is about to be inserted.  

software prefetching  
Software prefetching are prefetches that are coded by a programmer or implicitly created by the compiler.  

struct of array  
A struct containing or referring to arrays.  

vector field  
A collection of vectors that describe a flow in space.