A Performance and Scalability Analysis of the Tsunami Simulation EasyWave for Different Multi-Core Architectures and Programming Models

Steffen Christgau, Johannes Spazier, Bettina Schnor
Institute for Computational Science, University of Potsdam, Potsdam, Germany
email: {christgau, schnor, spazier}@cs.uni-potsdam.de

Berichte Institut für Informatik und Computational Science
UP, ISSN 0946-7580, TR-2015-1

A Performance and Scalability Analysis of the Tsunami Simulation EasyWave for Different Multi-Core Architectures and Programming Models

Steffen Christgau, Johannes Spazier, Bettina Schnor
Institute for Computational Science, University of Potsdam, Potsdam, Germany
email: {christgau, schnor, spazier}@cs.uni-potsdam.de

***Abstract*—In this paper, the performance and scalability of different multi-core systems is experimentally evaluated for the Tsunami simulation EasyWave. The target platforms include a standard Ivy Bridge Xeon processor, an Intel Xeon Phi accelerator card, and also a GPU. OpenMP, MPI and CUDA were used to parallelize the program to these platforms. The absolute performance of the application on the different platforms is compared, and limiting factors are analyzed based on the application's scaling behavior.**

***Keywords*-multi-core architectures, cache awareness, hybrid programming, scientific application, performance evaluation**

I. Introduction

Multi- and many-core systems, like multi-core CPUs, GPUs or accelerators enable applications to be executed fast if the application fits to the according hardware architecture. In this paper, we investigate the scaling of the stencil computation *EasyWave* [1] on different many-core systems. The application simulates the spatial propagation of a Tsunami wave that has been caused by a seismic event. The main purpose of EasyWave is its use in a Tsunami early warning center. Within such a center, sensor data is aggregated. In case of a seismic event, data such as estimations of the event's position and magnitude is used as input for EasyWave. In case EasyWave detects a hazard for a certain region, the operators of the warning center issue appropriate warnings to the government and people.

Due to the time critical character of the use-case, the application should run as fast as possible while providing accurate results to avoid false-positives as well as false-negatives. In prior work, it was shown that EasyWave can benefit significantly from the use of GPUs [2]. The originally sequential C++ application was ported to different generations of NVidia Tesla GPUs (Tesla and Fermi architecture) using CUDA and OpenACC. The runtime of a realistic single-run scenario was reduced from about five minutes to a few seconds, which is already acceptable for operators in early warning centers to decide on the situation and disseminate warnings.

In this paper, we compare the performance of EasyWave on different many-core architectures using also different parallel programming models: a parallel OpenMP version on a multi-core CPU as well as on the Xeon Phi and a single GPU using CUDA. On the Xeon Phi, offloading and native co-processor execution are examined. We compare both the absolute runtimes where applicable since this is the critical metric for the warning center operators, and the scalability of the application on the different architectures. The scalability results will help to derive implications for porting other applications to these platforms.

The remainder of this paper is organized as follows: In the next section, the experimental environments and their technical details are presented alongside a short description of the used programming model. EasyWave is presented in Section III. After that, the different employed parallelization techniques for EasyWave are explained in detail. Section V presents and discusses the experimental results. In Section VI, an overview of related work is given, followed by a conclusion and summary.

II. Environment and Programming Models

We tested the following three different platforms: A powerful conventional server system using a single Ivy Bridge Xeon processor, an Intel Xeon Phi 7120D, and a NVIDIA Tesla K40m GPGPU. A summary of the hard- and software configuration is shown in Table I.

A. Multi-core CPU

The multi-core CPU system is a conventional Intel Xeon server processor that consists of 10 cores. All cores share a last level cache and have private L2 and L1 which remain consistent due to the cache coherence protocol used in that processor. Further, the CPU's specification states it owns four memory channels and is able to deliver 59,7 GB/s of memory bandwidth [3]. Although the processor supports HyperThreading and TurboBoost, both technologies were not used respectively disabled to prevent disturbing the experimental results.

OpenMP [4] is used as framework to parallelize EasyWave. Among other frameworks such as POSIX pthreads, Cilk, or Intel Thread Building Blocks, OpenMP allows an easy yet efficient parallelization of the code. To compile the code, the Intel Compiler version 14.0.2 was used.

B. Intel Xeon Phi Co-processor

The Intel Xeon Phi 7120P (also known as *Knights Corner*) is a co-processor card attached to a host system via PCI-Express connection. It is based on the *Many Integrated Core* (MIC) architecture and consists of 61 processor cores that are derived from the original Intel Pentium architecture and execute program code in order. The cores are connected via a bidirectional ring which is realized by three rings for each direction: a data block ring, an address ring, and an acknowledgment ring. This bidirectional ring serves as means for memory transactions and maintains the coherence between the caches.

	Accelerator Machine		GPU machine	
property	Multi-Core CPU	Accelerator	GPU	Host
model	Intel Xeon E5-2690v2	Intel Xeon Phi 7120D	Tesla K40m	Intel Xeon E5-2690v2
frequency	3.0 GHz	1.24 GHz	746 MHz	3.0 GHz
cores/threads	10/20	61/244	2880 cores	10/20
memory	128 GB DDR3	16 GB GDDR5	12 GB GDDR5	64 GB DDR3
cache	32 KB L1	32 KB L1	48 KB L1	32 KB L1
	256 KB L2	512 KB L2	1536 KB L2	256 KB L2
	25 MB L3 (shared)	-	-	25 MB L3 (shared)
vector unit	256 b (AVX)	512 b	-	256 b (AVX)
interconnect	PCIe to Phi	on-chip rings	PCIe	InfiniBand
operating system	Linux 3.0.76	Linux 2.6.38.8	(none)	Linux 2.6.32
prog. model	OpenMP	OpenMP, MPI, hybr.	CUDA/SIMT	MPI

Table I: Hard- and software parameters of the programming environments.

As extension to the old Pentium processor, the cores possess a vector unit capable of handling 512 bits of data in parallel. Additionally, each core has support for four hardware threads (SMT). As one core is dedicated to the stream-lined Linux OS running on the SMP-like accelerator, a maximum number of 240 hardware threads can be used to execute an application. The L1 and L2 caches are dedicated to each core and are coherent among the chip [5]. The main memory on the accelerator comprises 16 GB.

For the co-processor, basically two different execution modes are available to run a program: First, the *native* mode that requires cross-compilation of the whole application (including dependencies, such as third-party libraries) and runs the whole application on the accelerator. To exploit the parallelism of the Xeon Phi, a parallelization has to be done. For this, implementations of well-known techniques like OpenMP or the Message Passing Interface (MPI) [6] are available. While the former requires compiler support, the latter requires an MPI implementation for the target platform. In the second execution mode, only some functions, most likely the compute-intensive ones, are compiled for and executed on the Phi. This mode is called *offload* and requires support by the compiler.

In our experiments, we used the Intel compiler Version 14.0.2 to compile EasyWave for the Xeon Phi. The tools that supported the development and execution of the generated code were included in the Intel C++ Studio XE1 2013 SP1. The MPI implementation we used was Intel MPI version 4.1.0.024.

C. Single GPU

In addition to the mentioned multi-core systems, an NVIDIA Tesla K40m GPU (Kepler architecture) is included in the comparison to evaluate the performance of a high-end GPU. As GPUs provide hardware support for computing massively parallel programs, the device class fits well to EasyWave. Previous work has shown that this kind of application can benefit from those devices [7], [2], and even from multiple GPUs [8] or clusters [9].

To program the GPU CUDA-C is employed: the CUDA toolkit version 5.5.22 and GCC version 4.8.2 were used. The code of the implemented kernels is a straight forward CUDA port of the according C++ methods from the sequential application with typical optimization considerations like ensured memory alignment.

III. EasyWave Application Details

The EasyWave tsunami simulation [1], [10] uses bathymetric data as input model for the ocean region to be simulated. The data is given as 2-dimensional regular grid. The computation carried out on the grid is repeated in a time-loop and divided into two parts: update of wave heights and update of reverse fluxes. Due to the employed numerical scheme the wave and flux updates require a three-point stencil, i. e. the update of a grid cell in a time step requires the current data of the grid cell and other data of two of its eight neighbors (Moore neighborhood). Additionally, the stencils differ in which cells are included in the computation. For wave update, the upper and left neighbors are used, whereas the flux update includes the lower and right neighbor cells. Note that the computation happens in place. No additional fields are allocated to store the new values, and no copying or flipping is done at the end of a single time-step.

The data required for the simulation are stored in separate arrays of single-precision floating points (structure of arrays). The input data is loaded at application startup from the file system.

For the wave height update step, nine memory accesses and seven floating point operations are necessary. For the flux update also nine memory accesses but six floating point operations are carried out. The index for the memory accesses is computed beforehand from the variables of the two loops iterating over the grid. Thus we assume that it is held in a register or at least in the cache hierarchy where the current stack frame including the local variables is very likely to be stored. As all data involve single precision floats of size 4 bytes, the operational intensity in FLOPs per loaded byte is $\frac{7\,\text{FLOPs}}{9\cdot 4\,\text{bytes}} = 0.194$ for wave height resp. 0.167 FLOPs/byte which is quite low. Thus, the application is clearly memory bound.

Although, the program allows storing the computed data in files for visualization and further processing purposes, this feature is ignored within this paper. Moreover, the sequential application uses a dynamic extension of the computed area, because of the limited area that needs to be updated/computed when the wave starts with its propagation. This feature is not supported in the parallelized versions. The lack of this optimization simplifies the parallelization.

IV. Application Parallelization

The parallelization approach of EasyWave for the different target platforms is discussed in the following subsections. For all described parallel implementations, we ensured that the computational results are still valid in comparison with the sequential version, i. e. only small deviations of the resulting wave height are allowed. The code was generally compiled with `-O3` as compiler option.

A. *Basic OpenMP parallelization*

As described in Section III, the application repeatedly iterates over a regular 2D grid for several time steps. Further, the computations of each grid point inside the two update procedures for wave height and fluxes are independent of each other. Using domain decomposition is therefore suitable to develop a parallel version of each of the two procedures. Thus, the compute domain is partitioned into one-dimensional chunks.

The partitions are created by annotating the vertical loop with a `parallel for` OpenMP pragma. This was done for both update procedures. This basic OpenMP variant of EasyWave was evaluated on the Xeon Phi as well as on the Xeon server processor.

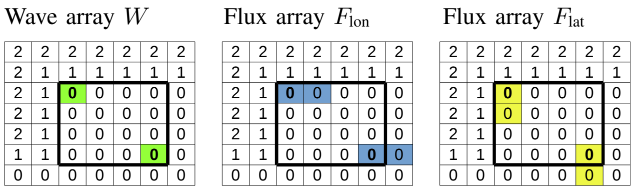
B. *Offloading*

The Xeon Phi's offloading execution mode was used in the *offload* version of the application. The wave and flux update functions were annotated to be offloaded and compiled for Xeon Phi, while the remainder of the software is executed by the host processor. Further, memory transfers to and from the accelerator were instructed to the compiler by adding according pragmas to the source code: Before the wave update and only in the first time-step, data is offloaded to the Xeon Phi and is finally copied back to the host system when the final time-step is completed. This minimizes the need for memory transfers over the (comparable slow) PCI-Express connection and only the invocation of the two update functions is issued from the host processor.

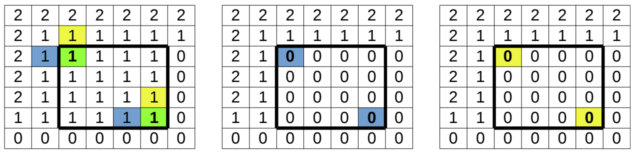
C. *Cache-aware OpenMP approach*

Typical data simulated by EasyWave has a size of some hundred MB. Thus, none of the target platforms is capable of holding the data in one of its caches. Even in the basic OpenMP version, each of the threads computes a sub-array that is larger than the caches. In addition, none of the computed or accessed elements is reused within one time step. Therefore, the utilization of the caches is quite bad. Further, the low operational intensity of the two update functions implies a high demand on memory bandwidth.

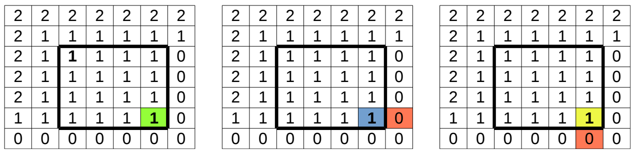
This motivated a parallel cache-aware variant of the basic OpenMP program. Within this approach, the compute domain is divided into smaller blocks that fit into the last level cache. These blocks are processed in sequence from left to right and from top to bottom. Each block is computed for multiple time-steps in parallel, using all the



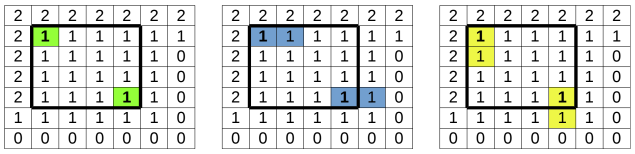
(a) 1. Iteration: Update of wave array.



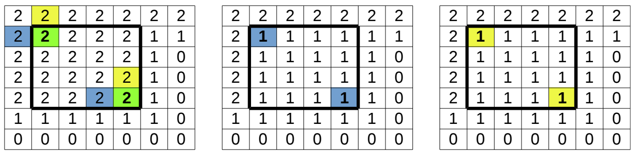
(b) 1. Iteration: Update of flux arrays.



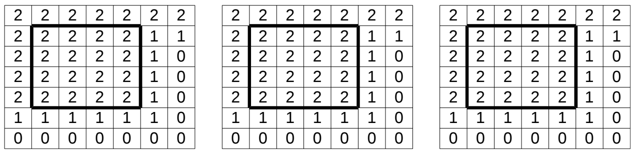
(c) Need of moving the block after the first iteration.



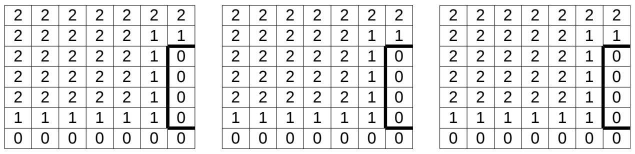
(d) 2. Iteration: Update of wave array.



(e) 2. Iteration: Update of flux arrays.



(f) Processing of the block completed.



(g) Continue with the next block in sequence.

Figure 1: Moving of the block area in the cache-aware OpenMP version.

available threads. This improves the cache usage, because the elements of a block are reused for multiple iterations.

Figure 1 demonstrates the processing of one such block in the middle of the two dimensional domain (highlighted with a bold border) and for two consecutive time-steps, although the algorithm applies for more timesteps as well. Three different arrays (W, F_{lat}, F_{lon}) are shown as they are part of the stencil. In the Figure, the arrays are represented as cells. The number inside a cell indicates the time step up to which its data has been computed so far. Because of the sequence in which the blocks are computed, it is guaranteed that all blocks to the left and to the top were already updated. They have a level greater than 0. Whereas, the outstanding blocks to the right and to the bottom are not computed yet.

At the beginning of the first iteration, the wave array $W(x, y)$ within the block is completely updated to the next level (i.e. to level 1) according to the formula[1]

$$\begin{aligned} W^{t+1}(x,y) = & W^t(x,y) \otimes \\ & F^t_{\text{lon}}(x,y) \otimes F^t_{\text{lon}}(x+1,y) \otimes \\ & F^t_{\text{lat}}(x,y) \otimes F^t_{\text{lat}}(x,y+1). \end{aligned} \tag{1}$$

This is possible because the elements needed from both flux arrays are available at the current level 0, as can be seen in Figure 1a. The required flux values are highlighted for two border elements of the wave array, showing the first stencil.

Afterwards all flux values are updated based on the newly computed wave elements of level 1 according to the simplified formulas

$$\begin{aligned} F^{t+1}_{\text{lon}}(x,y) = & F^t_{\text{lon}}(x,y) \otimes \\ & W^{t+1}(x,y) \otimes W^{t+1}_{\text{lon}}(x-1,y) \\ F^{t+1}_{\text{lat}}(x,y) = & F^t_{\text{lat}}(x,y) \otimes \\ & W^{t+1}(x,y) \otimes W^{t+1}_{\text{lat}}(x,y-1) \end{aligned} \tag{2}$$

As marked in Figure 1b, each flux element depends on two wave values according to the second stencil, which is reverse to the first one. The update of the longitudinal fluxes in the leftmost column is possible, because the elements outside the block were already calculated to level 1 in a previous block. The same applies to the latitudinal fluxes in the uppermost row.

The block has to be moved one column to the left and one row to the top, before the next iteration starts. The move is necessary because updating the rightmost column and the bottommost row of the wave array would otherwise result in inconsistent data. The reason is that some fields from the flux arrays, that are used within the computation, are now from an outdated level, as highlighted in red in Figure 1c. A positive effect of moving the block is that fields from neighbor blocks, which could previously not be updated to the latest level, because of the move mechanism itself, are now processed further.

After moving the block, the second iteration can be accomplished analogously to the first one as illustrated in Figure 1d and 1e. As a result, the wave and flux values of the current block location have been updated to level 2 (see Figure 1f). Hence, the local update within the block is completed and the next block from the sequence can be processed according to the same pattern (Figure 1g). If all blocks are updated, the entire algorithm is repeated until the desired number of iterations is reached.

The advantage of this cache-aware algorithm is that no redundant computations are necessary. Each array element is processed exactly as often as required to update to a desired iteration. This is mainly achieved by the block moving scheme which exploits the structure of the underlying stencils. In general, the algorithm cannot easily be applied without adaptations. Other stencils may require redundant calculations and additional memory buffers. However, even a higher computational effort can pay off, if the architectural limitations can thus be bypassed.

Although this optimization is cache-aware it implies higher coding efforts and complicates the source code of the application. Additional execution overhead is produced as more loops and computations for loop ranges and block boundaries are required to enable the cache-aware block computation.

D. Vectorization

Since EasyWave computes on adjacent elements of its data grid, the application may benefit from vector instructions. Therefore, a compiler-based vectorization was used on the two target platforms which offer vector instructions. Hand-written vectorized codes using intrinsics were also evaluated but were not faster than compiler generated ones.

The compiler-based vectorization was finally used on the two target platforms which offer vector instructions, i.e. the Xeon and the Xeon Phi. The vectorization was applied both to the basic and the cache-aware OpenMP version of EasyWave. Although supported by the compiler, the OpenMP SIMD pragma was not used, as preceding work indicates slower performance as native compiler-based vectorization [11]. For application versions without vectorization presented in this paper, the compiler's `-novec` option was used to explicitely disable vectorization in the `-O3` optimization level.

E. Message Passing

In the message passing implementation of EasyWave, we used MPI to parallelize the program. The used approach is well-known: First, the boundaries of a process' subdomain are computed. Next, the updated boundary values are transferred to the neighboring processes. To have a potential for overlapping communication and computation, non-blocking routines (`MPI_Isend/Irecv`) are utilized to perform the data exchange: The transfer is initialized and might be completed by the MPI implementation while the application is computing the inner parts of the according subdomain. After the calculation, the application waits for the data transfer to be completed using `MPI_Wait`.

[1]$\otimes$ is a placeholder for a floating point operation

V. Experimental Results

The absolute runtimes as well as the speedup will be discussed in the following. The runtimes are important for the use case scenario in an early warning center where the result has to be calculated as fast as possible. To investigate the scaling of the application on the different platforms, we analyze the speedup which is each time calculated regarding the sequential run of the corresponding parallel program (i.e. regarding the sequential basic version or the sequential cache-aware version).

For all runtime values, the minimal value of five repeated measurements is reported. They do not include the time required to load the data from disk into memory. The data set used for the multi-core CPU and the Xeon Phi are equal.

If not stated otherwise, a grid of size 2701×2446 represents a region of the Indian Ocean with nearly no mainland. This prevents computational imbalance between the processes introduced by the data, as no computation is done on land areas by EasyWave.

A. Multi-Core CPU

The runtimes on the Ivy Bridge processor are shown in Figure 2 for the basic OpenMP version and its cache-optimized counterpart, each with and without vectorization used. For every version, scaling up to about six cores can be observed, but the versions differ significantly in runtime.

The runtime of the basic version decreases until seven cores are used. Afterwards, the speed of the application remains constant. The same applies to the vectorized version, but this one shows better runtimes for lower processor counts. This indicates that the memory intensive characteristics of EasyWave impose too much load on the memory controllers. This was verified by using the Intel Performance Counter Monitor. We observed a total memory bandwidth of 47 GB/s used by both the vectorized and the unoptimized basic OpenMP version. This measured limit is 78 % of the theoretical maximum bandwidth. For the vectorized version, this limit is already reached for five cores. From this, we can conclude that vectorization can increase the performance, but only as long as the memory controller is no bottleneck. In summary, vectorization shows no benefit using the full system (10 cores).

In case of the cache-aware version, the runtime decreases significantly using vectorization. Figure 2 shows that only by exploiting the two technologies, the application performance can be tuned significantly. It can be seen that the cache-aware version without vectorization is slower even than the basic version for 1 up to 4 cores. This is due to additional overhead/statements introduced by the cache-awareness algorithm. But for the interesting case, the full system, the benefit of parallelization by vectorization is significant, since due to the code optimization the memory controller can keep up.

The effects described above have influence on the scaling of the application as shown in Figure 3. The speedup values are reported in relation to the single-thread

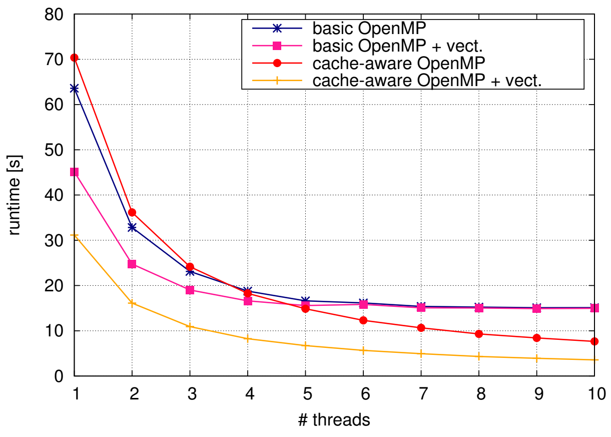


Figure 2: EasyWave runtimes on the multi-core Xeon CPU for the basic and the cache-aware versions including vectorization.

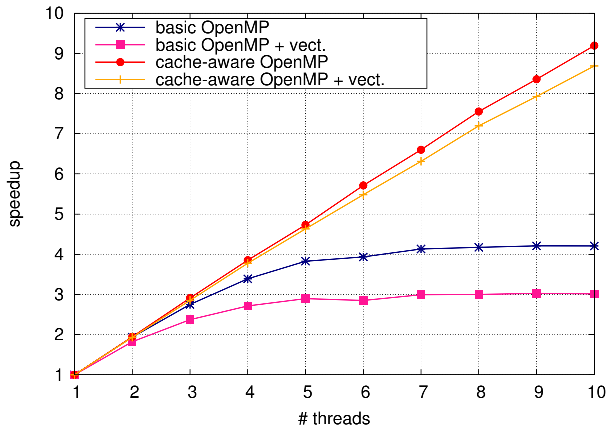


Figure 3: Speedup of EasyWave on the multi-core Xeon CPU.

runtime of the according program version. For the non-cache-aware variants, the memory bandwidth limitation clearly limits further scaling, whereas the cache-optimized versions show a much better scaling behaviour which is nearly linear. Thus, with appropriate coding effort, the application can scale with core count, but the architectural limits have to be taken into account.

B. Intel Xeon Phi Co-processor

For the Intel Xeon Phi, the different execution modes were analyzed first. We compared the performance of the basic OpenMP version against the offload variant, where only the two update functions are executed on the accelerator (cf. Section IV-B). In the direct comparison, the offload version always runs slower than its native counterpart. While the basic version took 323.4 s with one thread, the offload variant used 331.1 s, which equals an overhead of 2.4 %. When the maximum number of 240 threads is used, the basic version runs 6.98 s, where the offload version needs 8.17 s (+17 %). The overhead can be attributed to the frequent startups of the offloaded functions and the implied thread start, which should be avoided. In consequence, the offload mode is not considered in the remaining parts of this work.

Although the offload mode seems to be disadvantageous for EasyWave, it might be beneficial for applications that can overlap certain tasks with the computation on the Xeon Phi. For example, simulations which require intermediate results to be recorded (e. g. for visualization purposes) could run the computation on the Phi for some iterations, then copy the memory back to the host. While the Phi continues to compute in an asynchronous function call, the host CPU can save the results on local disk which is likely to be much faster than using NFS from the Xeon Phi.

In Figure 4, the absolute runtimes on the Intel Xeon Phi are presented on an logarithmic axis for an increasing number of threads. For up to 60 threads, no SMT is in use as we take care that threads are distributed among cores.

Comparing the base OpenMP version with and without vectorization, Figure 4 shows that vectorization has a very positive effect on the runtime: 17.45s versus 8.55s for 60 threads, and 8.72s versus 6.98s for 240 threads. This is different from the situation on the multi-core CPU. Although, the theoretical 16× performance improvement (sixteen 32-bit floats can be held by one vector register) is not achieved, due to the low operational density of the application.

Also different to the Ivy Bridge Xeon CPU, the cache-aware, non-vectorized OpenMP version is always slower than the basic OpenMP one. There is no inflection point where the overhead of being cache-aware finally pays off as it did after four cores on the Ivy Bridge. This can be attributed to the less sophisticated design of the Phi's cores, which execute instructions in order, do not possess a branch predictor nor do they run at high frequencies.

A real performance boost brings the cache-aware OpenMP version with vectorization. This version outperforms all other program variants on the Xeon Phi at all core counts.

Looking at thread counts larger than 60, the runtime for all versions only decreases slightly. In case of the base version with vectorization, the runtime is 8.55s on 60 cores versus 6.72s on 180 cores and increases again to 6.98s for 240 cores. The corresponding figures for the cache-aware OpenMP version with vectorization are 4.19s (60), 4.25s (180), and 3.61s (240). Thus, SMT is not much beneficial for this application, i. e. only small speedup improvements are achieved as soon as the vector unit is used.

This is also visible in Figure 5 where the speedups relative to their single thread runs are shown. Looking on the scaling of these versions, only for the two non-vectorized variants the speedups still increase. This is consistent with the observations in [11] where the memory controller delivers more performance when SMT is used. In case of the memory-bound application EasyWave, we can therefore observe an slightly increasing speedup. Contrary, the speedup of all other versions is saturated after 60 cores. Further, SMT does not bring as much performance improvements for the observed application use-case as vectorization did.

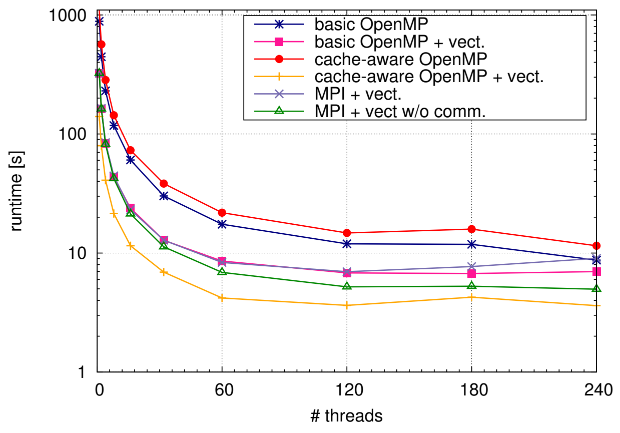


Figure 4: EasyWave runtimes on the Intel Xeon Phi.

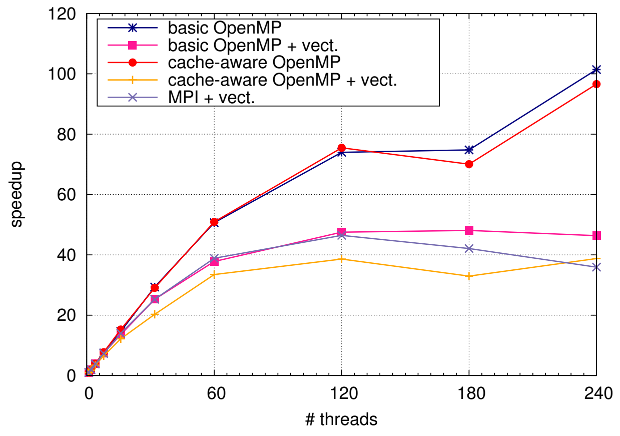


Figure 5: Speedup of EasyWave version on the Intel Xeon Phi.

For MPI, the absolute runtimes are also shown in Figure 4. We report the timings for an MPI version that was compiled to use the vector units but without cache-awareness or other OpenMP parallelization. Up to 120 cores, its performance is nearly identical to the vectorized OpenMP version. After that, the runtime increases significantly and is finally even slower than the basic OpenMP implementation. This can be attributed to the communication, as the same MPI program without communication (shown in the green/triangle curve) does not show such an increase in runtime. Note that this version produces wrong results, but it was used to analyze the influence of the communication overhead.

In addition to the vectorized MPI version, a hybrid version was analyzed that uses one MPI process per core and two respectively four OpenMP threads to calculate the process' subdomain. In the experiments, no performance gain was achieved.

C. *Final Comparison and Discussion*

The most minimum runtime on the Xeon Phi (3.61 s) is still higher than for the Ivy Bridge Xeon (3.5 s) for the same input data. Thus, purchasing and porting the accelerator has no benefit for EasyWave and other applications of its class with respect to runtime. Moreover, the parallel efficiency (speedup/thread count) is quite worse on the Xeon Phi compared to the Ivy Bridge Xeon. Here, the fastest version (cache-aware and vectorized OpenMP) has an efficiency of 87 % while the same (and also fastest) version on the Phi only has a efficiency of 16 % when using 240 cores (32 % efficiency for 120 cores with approx. equal runtime).

Finally, we compare the absolute runtimes of EasyWave on the multi-core Xeon processor, the Xeon Phi, and on a single GPU. To measure the benefit for the operation in an Early Warning Center, the speedup related to the original sequential CPU application is reported. For all systems, the experiment is conducted using the same input data set which is larger than the one used in the previous chapters. The grid is extended to a size of 18001×16301 cells but represents the same area of the Indian Ocean with higher resolution as in the previous experiments. The reported numbers are the minimal runtimes of the parallel application version on the according platform.

In all presented cases a significant improvement, i. e. speedup, over the sequential version can be observed as shown in Table II. The positive effect of vectorization can be seen on both architectures, the Xeon server CPU (speedup of 3.6) and the Xeon Phi (speedup of 11.9).

Again, cache-awareness pays out a lot for both architectures. The hand-coded cache-optimized version on the Ivy Bridge Xeon performs best and its runtime is 20 % lower than the next best variant, the cache-aware and vectorized version on the Xeon Phi.

While the GPU can compete with the vectorized Phi version, it is clearly outperformed by the cache-aware and vectorized version on the Xeon Phi with the runtime being 150 s lower. So, the additional effort for the CUDA port (i.e. recode the application to that programming model) does not pay off for EasyWave.

These results might question the need for an accelerator like the Xeon Phi or a GPU as the commodity (yet expensive) server processor shows the best performance. However, it should be noted that the results might only be valid for the presented application, although it is a typical representative of its application class. Further, the best performance was achieved by according programming skills. In contrast, the vectorized version for the Xeon Phi was created only with the compiler's help and it is already 3.3 times faster than the according server CPU version. With that in mind, the Xeon Phi can massively improve an application's performance without further coding efforts and/or the required time and skills of a programmer. Yet, optimizing to the according hardware platform is still the key to achieve the maximum of performance.

VI. Related Work

In this section, we summarize related work which presents experiences with porting different types of applications to the Intel Xeon Phi.

system	program version	runtime	speedup
Ivy Bridge Xeon	sequential	5593 s	-
Ivy Bridge Xeon	vectorized	1577 s	3.6
Ivy Bridge Xeon	cache-aware + vect.	268 s	20.9
Xeon Phi	vectorized	471 s	11.9
Xeon Phi	cache-aware + vect.	331 s	16.9
single K40m	CUDA port	482 s	11.6

Table II: Overall comparison of parallel EasyWave versions among the different systems.

The work presented in [12], [13] shows the importance to interpret measurement results always within the investigated application class. Cramer et al. compare the performance of a Xeon Phi prototype card possessing 60 cores/240 hardware threads with a massive 128-cores SMP computer [12]. The performance and scalability of a cache-friendly version of a conjugate gradient (CG) solver is discussed in detail. Their experiments show that the considered CG application shows a better scaling behaviour on the Xeon Phi co-processor than EasyWave. Without SMT (i.e. 60 cores), a speedup of over 53 is observed and with all 244 hardware threads in use the speedup increases up to over 74. Different to our experiments, they observed that the gain from vectorization within the CG method is quite small on the Xeon Phi.

In the successor paper, Schmidl et al. [13] extend their work and analyze the performance of several other scientific applications ranging from the NPB package to applications of their university coming from different science domains. Further, their performance is compared with a commodity two-socket Intel *Ivy Bridge* E5 Xeon processor system, a predecessor of the multi-core CPU used in our experiments. It is shown that the speedup on the Xeon Phi is good, but compared to the Ivy Bridge system, the absolute performance is lower even if the full accelerator core capabilities are used. This is again different from our experience where the basic OpenMP version was outperformed by the Xeon Phi. The authors assume that the benchmarks suffer from the slow sequential performance of the Phi's in-order cores and higher memory latency.

The authors of [14] explore how popular programming models behave on a modern processor like the Xeon Phi when multi-programming is used. To answer this question, three benchmarks (Fibonacci computation, mergesort, dense matrix multiplication) from different application categories are executed first exclusively and then together on a Xeon Phi 5110P. As programming models, OpenMP, Cilk and Thread Building Blocks (TBB) are used. From the exclusive runs, the scalability of the three benchmarks can be observed. In all three cases a saturation in the measured speedup can be observed for every programming model.

Pennycook et al. [15] optimize the miniMD benchmark, a molecular dynamics application, for better use of SIMD. They focus on the SIMD instruction allowing scatter and gather operations as well as masking operations. They compare the performance of their new optimized implementation on an 8-core Intel Xeon E5 and on a Xeon Phi co-processor. In the outcome, the Xeon Phi is up to 42 % faster than its counterpart. The authors highlight the need to optimize the code and ensure that SIMD is used effectively.

In [11], a further application class, a medical CT image reconstruction is discussed and the performance on a two-socket Xeon E5-2660, a Tesla K20 (Kepler architecture) and a Xeon Phi 5110P is compared. As in [15], the authors exploit gather and SIMD operations to tune their application for the Xeon Phi. The experiences for this application show that only hand-tuned and well-designed assembly code for the Xeon Phi can produce performance that is superior to the Xeon E5-2660. They demonstrate the limits of the memory controller, since only 50 % of the theoretical peak memory bandwidth could be obtained for an updating kernel used by their application. They also point out that this is only possible when using the Phi's SMT capabilities to a maximum. Not surprisingly, the GPUs deliver best performance for the examined application as its bi-linear interpolation part can benefit from the hardware support provided by the texture buffers.

VII. Conclusion

In the presented work, the absolute performance and the scalability of the simulation application EasyWave was evaluated on different multi-core processor systems. It was shown that for this memory-bound application, tuning towards this is a critical aspect on a standard Xeon processor as well as on the Xeon Phi.

Influence of vectorization: For the Ivy Bridge Xeon processor, comparing the base OpenMP version with and without vectorization vectorization shows no benefit for the interesting case of the maximum number of ten cores since the memory controllers cannot keep up.

The situation is different on the Xeon Phi. Comparing the base OpenMP version with and without vectorization, has shown that vectorization has a very positive effect on the runtime of EasyWave on the Xeon Phi.

Influence of SMT: In case of EasyWave, SMT is not much beneficial for this application, i. e. only small speedup improvements are achieved as soon as the vector unit is used.

Comparing programming models: We were interested to compare the different available programming models on the Xeon Phi. But the investigated programming models OpenMP, MPI, and the hybrid approach combining MPI and OpenMP show very similar runtimes. For the MPI version it is recommended to use not more than 120 threads, since otherwise the communication overhead engulfs any benefit due to SMT.

Human versus hardware effort: Using a cache-aware algorithm, the application performs best on the standard Xeon and beats the Xeon Phi accelerator card as well as a single GPU thanks to its faster cores.

But without any manual tuning effort, the basic OpenMP version of EasyWave gets a performance improvement on the Intel Xeon Phi: The fastest runtime on the multi-core CPU is 15.1 s which is reduced to 6.98 s using the Xeon Phi.

References

[1] A. Babeyko, "EasyWave: Fast Tsunami Simulation Tool for Early Warning," February 2012, ftp://ftp.gfz-potsdam.de/pub/home/mod/babeyko/easyWave/easyWave_About.pdf.

[2] S. Christgau, J. Spazier, B. Schnor, M. Hammitzsch, A. Babeyko, and J. Waechter, "A comparison of CUDA and OpenACC: Accelerating the tsunami simulation EasyWave," in *ARCS 2014 - 27th International Conference on Architecture of Computing Systems, Workshop Proceedings, February 25-28, 2014, Luebeck, Germany*. VDE Verlag / IEEE Xplore, 2014.

[3] Intel ARK, "Intel Xeon Processor E5-2690 v2 specification." [Online]. Available: http://ark.intel.com/products/75279/Intel-Xeon-Processor-E5-2690-v2-25M-Cache-3_00-GHz

[4] OpenMP Architecture Review Board, "OpenMP application program interface version 4.0," Jul. 2013. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[5] J. Reinders, "An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors," Nov. 2012. [Online]. Available: https://software.intel.com/sites/default/files/article/330164/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors_1.pdf

[6] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 3.0*. High Performance Computing Center Stuttgart, Sep. 2012.

[7] M. T. Satria, B. Huang, T.-J. Hsieh, Y.-L. Chang, and W.-Y. Liang, "GPU acceleration of tsunami propagation model," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 5, no. 3, pp. 1014–1023, June 2012.

[8] M. L. Sætra and A. R. Brodtkorb, "Shallow water simulations on multiple GPUs," in *Proceedings of the 10th International Conference on Applied Parallel and Scientific Computing - Volume 2*, ser. PARA'10. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 56–66. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28145-7_6

[9] D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michea, "High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster," *Journal of Computational Physics*, vol. 229, no. 20, pp. 7692–7714, 2010.

[10] German Research Centre for Geosciences, "EasyWave," online, 2014, http://trac.gfz-potsdam.de/easywave.

[11] J. Hofmann, J. Treibig, G. Hager, and G. Wellein, "Performance engineering for a medical imaging application on the Intel Xeon Phi accelerator," in *ARCS 2014 - 27th International Conference on Architecture of Computing Systems, Workshop Proceedings, February 25-28, 2014, Luebeck, Germany*, W. Stechele and T. Wild, Eds. VDE Verlag / IEEE Xplore, 2014.

[12] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey, "OpenMP programming on Intel Xeon Phi coprocessors: An early performance comparison," in *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, Nov. 2012, pp. 38–44.

[13] D. Schmidl, T. Cramer, S. Wienke, C. Terboven, and M. S. Müller, "Assessing the performance of OpenMP programs on the Intel Xeon Phi," in *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, F. Wolf, B. Mohr, and D. an Mey, Eds. Springer, 2013, pp. 547–558. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40047-6_56

[14] A. Tousimojarad and W. Vanderbauwhede, "Comparison of Three Popular Parallel Programming Models on the Intel Xeon Phi," in *Euro-Par 2014: Parallel Processing - 20th International Conference, Porto, Portugal, August 24-28, 2013. Proceedings*. Springer, 2014.

[15] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis, "Exploring SIMD for molecular dynamics, using Intel Xeon processors and Intel Xeon Phi coprocessors," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1085–1097. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2013.44