EasyChair Preprint

№ 277

Task-Based Programming on Emerging Parallel Architectures for Finite-Differences Seismic Numerical Kernel

Salli Moustafa, Wilfried Kirschenmann, Fabrice Dupros and Hideo Aochi

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

June 18, 2018

Task-Based Programming on Emerging Parallel Architectures for Finite-Differences Seismic Numerical Kernel

Salli Moustafa[1], Wilfried Kirschenmann[1], Fabrice Dupros[2], and Hideo Aochi[2]

[1] ANEO, Boulogne-Billancourt, France
{s.moustafa,w.kirschenmann}@aneo.fr
[2] BRGM, Orléans, France
{f.dupros,h.aochi}@brgm.fr

Abstract. In recent years, heterogeneous hardware have generalized in almost all supercomputer nodes, requiring a profound shift on the way numerical applications are implemented. This paper, illustrates the design and implementation of a seismic wave propagation simulator, based on the finite-differences numerical scheme, and specifically tailored for such massively parallel hardware infrastructures. The application dataflow is built on top of PaRSEC, a generic task-based runtime system. The numerical kernels, designed for maximizing data reuse can efficiently leverage large SIMD units available in modern CPU cores. A strong scalability study on a cluster of Intel KNL processors illustrates the application performances.

Keywords: High-Performance Computing, C++ generic programming, SIMD, Task-based runtime system, PaRSEC, Seismic Wave Propagation

1 Introduction

Since the advent of multicore processor at the beginning of 2000's, the compute power is not more driven by the clock frequency [1], but instead, by the number of functional units into multi/many-core processors featuring large SIMD units or in accelerators such as GPUs. The combination of these computing devices lead to highly heterogeneous hardware infrastructures. Hence, maximizing application performance on such systems is a major challenge [2–4]. Indeed, for coping with these systems, application developers use to mix several programming paradigms, following the now classical MPI+X approach. MPI manages communication through the network interconnect and is completed by OpenMP, Intel TBB, CUDA or OpenCL for addressing each node.

Task-based approach coupled with a generic runtime system is an emerging programming paradigm that greatly improves programmer productivity, leaving him to focus on the algorithm and computational kernels implementation. From this perspective, building high-performance codes require fine-tuned kernels. As the main bottleneck on modern platforms is the memory bandwidth [5], those

kernels must ensure a good data locality. In addition, the kernels have to leverage the SIMD units available on modern processors. To achieve both performances and portability across various architectures, the kernel must use generic high-level concepts (like a DSL[3]) encapsulating these specific optimizations [6].

In this work, we conducted a study on the above mentioned challenges through the linear seismic wave propagation problem. Seismic wave propagation from an earthquake in the Earth has been always numerical challenge, because of its dimension, resolution and medium complexity with respect to the geoscientific knowledge and engineering requirements (e.g. [7] and [8]). Basic problem is well formulated in the framework of elasto-dynamic equations and linear elasticity. Finite difference method has been applied since 1970-80's (e.g. [9]), suitable for structural discretization of continuous medium. In particular, the $4th$ order approximation in space on staggered grids is the most popular option because of its efficiency and stability (e.g. [10] and [11]).

As reported in several recent research papers (e.g. [12, 13]), explicit parallel elastodynamics application usually exhibits very good weak and strong scaling up to several tens of thousands of cores. Significant works have been made to extend this parallel results on heterogeneous and low-power processor (e.g. [14,15]). For instance the efforts to benefits from modern architecture with large vector unit is described in [16] where the use of explicit intrinsics appears mandatory to squeeze the maximum performance out of the underlying architecture.

One major contributions of this paper is the design and implementation of a fully task-based model for the seismic wave propagation into the SeWaS application [17]. To the best of our knowledge, this is the first end-to-end task-based implementation of the seismic wave model including the time-step dependency and efficient vectorization. We validate our results with the ONDES3D [18] production code. We consider Intel KNL manycore platforms for our experiments because of the complexity of such architecture and its capability to deliver both high memory bandwidth level and high peak floating point performance thanks to its AVX-512 units.

This paper is organized as follows. Section 2 provides the numerical background on seismic wave propagation. The task-based algorithm is described in Section 3 and its implementation in Section 4. Finally, we discuss the parallel performances in Section 5 and Section 6 concludes this paper.

2 Numerical Background and Classical implementation

2.1 Numerical scheme

Let us recall that elasto-dynamic equations allow for evaluating the three components of the velocity (V_x, V_y, V_z) and stress field (σ_{xx}, σ_{yy}, σ_{zz}, σ_{xy}, σ_{xz}, σ_{yz}) of the seismic wave. Fully discretized forms of these quantities as given in [10]. For instance, the discretized form of V_x is given in (1).

$$V_x^{t+\frac{1}{2}}(i+\frac{1}{2},j,k) = V_x^{t-\frac{1}{2}}(i+\frac{1}{2},j,k)+(D_x\sigma_{xx}+D_y\sigma_{xy}+D_z\sigma_{xz})^t(i+\frac{1}{2},j,k) \quad (1)$$

[3] Domain Specific Language

Parameter	Definition	Possible values
d_s	Spatial discretization step (km)	> 0
(l_x, l_y, l_z)	Size of the global domain (km)	$(> 0, > 0, > 0)$
(n_x, n_y, n_z)	Total number of grid points	$(l_x/ds,\ l_y/ds,\ l_z/ds)$
$T_{\max}$	Duration of the simulation (s)	> 0
dt	Time step (s)	> 0
N_t	Number of time-steps	$2 * T_{\max}/dt$
(c_x, c_y, c_z)	Number of grid points per tile	$(> 0,> 0,> 0)$
(n_{xx},n_{yy},n_{zz})	Total number of tiles per axis	$(n_x/c_x,\ n_y/c_y,\ n_z/c_z)$
t	Time-step index	$\{1, \cdots, N_t - 1\}$
l	Location of the halo within a tile	$\{0, \cdots, 5\}$
d	velocity component	$\{0, 1, 2\}$
s	Stress field component	$\{0, \cdots, 5\}$
(ii, jj, kk)	Coordinates of a tile	$[\![0, n_{xx}-1]\!] \times [\![0, n_{yy}-1]\!] \times [\![0, n_{zz}-1]\!]$
(i, j, k)	Coordinates of a cell within a tile	$[\![0, n_x - 1]\!] \times [\![0, n_y - 1]\!] \times [\![0, n_z - 1]\!]$

Table 1: Notations used to define tasks.

D_x is the $4th$ order central finite difference operator defined in (2).

$$\begin{aligned} D_x f(x, y, z) = c_1 * f(x + \Delta_1, y, z) + c_2 * f(x - \Delta_2, y, z) \\ + c_3 * f(x + \Delta_3, y, z) + c_4 * f(x - \Delta_4, y, z), \end{aligned} \tag{2}$$

where c_i and Δ_i, $i = 1, \cdots, 4$, are constant numerical coefficients depending on the discretization scheme. D_y and D_z are defined similarly. We considered this numerical scheme as a basis for building our seismic wave simulator for modern architectures. In particular, we designed a novel task-based implementation of the seismic wave propagation and implemented generic performance-portable numerical kernels.

2.2 Standard implementation

In the remaining of the paper, we assume the notations defined in Table 1. The general algorithm describing the seismic wave propagation is given in Algorithm 1. The model takes as input the velocities of the compression and shear waves, V_p and V_s, defined for every layer contained within the computational domain; the density ρ and the seismic sources. Given the inputs, the algorithm iterates over all time-steps and successively computes the velocity components (Line 7) and stress field components (Line 10) for every spatial grid point.

The velocity and the stress fields are evaluated at grid points separated by half of the discretization step. This is coming from the implementation of the staggered grid described for instance in [9]. Velocity computation at time step t depends on some computed values of the stress field at time step $t - 1$, as shown in (1). These dependencies represent the *stencil* for the computation of the velocity and are illustrated in Figure 1. Similarly, the computation of the stress

Algorithm 1: Linear seismic wave propagation

```
   In  : V_p, V_s, ρ, sources
   Out: Global Velocity and Stress of the seismic wave
 1 for t ∈ {0, 2, ··· , N_t − 2} do
 2   for i ∈ {0, ··· , n_x − 1} do
 3     for j ∈ {0, ··· , n_y − 1} do
 4       for k ∈ {0, ··· , n_z − 1} do
 5         ▷ Compute V_x, V_y and V_z
 6         for d ∈ {x, y, z} do
 7           ComputeVelocity(d, t, i, j, k);
 8         ▷ Compute S_xx, S_yy, S_zz, S_xy, S_xz and S_yz
 9         for s ∈ {xx, yy, zz, xy, xz, yz} do
10           ComputeStress(s, t + 1, i, j, k);
```

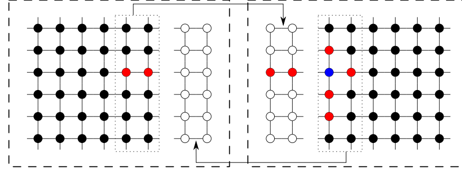


Fig. 1: Stencil for V_x computation (2D) and halo exchange between subdomains. Each subdomain, delimited by dashed boxes, is enlarged with the additional points for containing the halo. Computing the velocity at the blue colored grid point requires computing the stress fields at the red colored grid points.

field depends on some previously computed values of the velocity. Nevertheless, computing the different velocity components are independent and the same for all the stress field components. Hence, each of the loops at Line 6 and Line 9 can be evaluated in parallel.

The parallelization of this algorithm on this distributed memory computers has been extensively studied, relying mainly on MPI and OpenMP. Typically, the time-step loop (Line 1) is considered as sequential and the parallelization is achieved by regularly splitting the spatial domain among all the workers. Each MPI process will handle $(n_x/P) \times (n_y/Q)$ cells, where P and Q are the total number of MPI processes along the x and y axes. Within each MPI process, a team of OpenMP threads is spawned to perform the computations on the local cells. At the end of each time-step, each MPI process exchanges its boundaries with neighboring processes as depicted in Figure 1. For instance, a classical implementation is described in [18]. However, there are three main concerns with this approach: explicit synchronization required when exchanging the halo at the end of each time-step; serialization of the time-step loop and the lack of explicit vectorization.

3 A Fully task-based Model of the Linear Seismic Kernel

We considered the Algorithm 1 as a basis, and we made several optimizations on it, with the goal of making it scalable at high core count : we redesigned the algorithm into a fully task-based version including the time-step loop. Throughout this process, we considered different constraints: scalability, communication overlap, data locality, and vectorization efficiency.

Scalability : A task is an independent unit of work that will be processed by a thread. For instance, the computation of the x component of the velocity (rest. stress field) at time t and on the cell (i, j, k) is a task. To scale at high-core count, the algorithm must expose a large number of such tasks that can be processed concurrently. To that end, we need to define dependencies that are going to be used by a runtime system. It will schedule the tasks as soon as they become ready among all available computing resources. An important point to note is that the runtime system introduces some scheduling overhead due to calculations required for selecting the task to execute. To enhance the scalability of our model, by minimizing the former overhead, we partitioned the global domain into a set of 3D *tiles* defined by a contiguous collection of cells, on which the tasks are going to work, rather than on a single cell. In the following, we will omit the prefix 3D when referring to a tile, and we will use the index (ii, jj, kk) for its coordinates. Due to the nature of the stencil computation scheme, each tile will be enlarged to contain the halo retrieved from neighboring tiles, as in Figure 1.

Instead of performing the extraction and update of the halo, we add two new tasks dedicated to these actions: ExtractVelocityHalo and UpdateVelocity with the advantage of increasing the amount of available parallelism. This is required to maximize the parallel efficiency on modern distributed computing platforms. In summary, our task-based algorithm contains six types of tasks:

1. ComputeVelocity(d, t, ii, jj, kk) computes the d component of the velocity on tile (ii, jj, kk) at time-step t. It depends on UpdateStress$(s, t-1, ii, jj, kk)$.
2. ExtractVelocityHalo(d, t, ii, jj, kk) first extracts all the boundaries of the d-component of the velocity on the tile (ii, jj, kk). The extracted boundaries are placed in temporary buffers and sent to the UpdateVelocity tasks on neighboring tiles. It depends on ComputeVelocity(t, d, ii, jj, kk).
3. UpdateVelocity(d, t, ii, jj, kk) receives the velocity halo from neighbouring ExtractVelocityHalo tasks and update the tile (ii, jj, kk). It depends on ExtractVelocityHalo(d, t, ii, jj, kk).
4. ComputeStress(s, t, ii, jj, kk) defined similarly as ComputeVelocity.
5. ExtractStressHalo(s, t, ii, jj, kk) defined similarly as ExtractVelocityHalo.
6. UpdateStress(s, t, ii, jj, kk) defined similarly as UpdateVelocity.

The fully task-based version of the linear seismic wave propagation is presented in Algorithm 2. In Figure 2, we give the data-flow corresponding to this algorithm. On the same tile, and for a fixed d and t, the tasks ComputeVelocity, ExtractVelocityHalo and UpdateVelocity are serialized. As we will see

Algorithm 2: Task-based algorithm for the seismic wave propagation

In : V_p, V_s, ρ
Out: Global Velocity and Stress of the seismic wave

```
1  forall t ∈ {0, 2, ··· , N_t − 2} do
2      ▷ Compute V_x, V_y and V_z
3      forall d ∈ {x, y, z} do
4          forall ii ∈ {0, ··· , n_xx − 1} do
5              forall jj ∈ {0, ··· , n_yy − 1} do
6                  forall kk ∈ {0, ··· , n_zz − 1} do
7                      ComputeVelocity(d, t, ii, jj, kk);
8                      ExtractVelocityHalo(d, t, ii, jj, kk);
9                      UpdateVelocity(d, t, ii, jj, kk);
10     ▷ Compute S_xx, S_yy, S_zz, S_xy, S_xz and S_yz
11     forall s ∈ {xx, yy, zz, xy, xz, yz} do
12         forall ii ∈ {0, ··· , n_xx − 1} do
13             forall jj ∈ {0, ··· , n_yy − 1} do
14                 forall kk ∈ {0, ··· , n_zz − 1} do
15                     ComputeStress(s, t + 1, ii, jj, kk);
16                     ExtractStressHalo(s, t + 1, ii, jj, kk);
17                     UpdateStress(s, t + 1, ii, jj, kk);
```

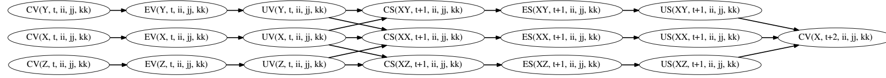


Fig. 2: Data-flow for the task-based linear seismic wave model (case of V_x computation). CV, EV and UV designate ComputeVelocity, ExtractVelocityHalo and UpdateVelocity. A similar definition holds for CS, ES and US.

in Section 5, combining the time-step loop with the spatial cells when defining tasks allows overlapping computations of different time-steps, hence reducing the execution time.

Communication overlap: by delegating extraction and update of the halo per tile to dedicated tasks, we enforce the potential of overlapping communications with computations. Indeed, once a ComputeVelocity task is completed, we can start extracting and sending its boundaries to UpdateVelocity tasks associated to neighboring tiles, while another ComputeVelocity task is in progress.

Data locality: to maximize cached data reuse, we considered a hierarchical representation of the manipulated data structures, typically V and S, as following:

– $V(x|y|z)(ii, jj, kk)(i, j, k)$

– $S(xx|yy|zz|xy|xz|yz)(ii, jj, kk)(i, j, k)$

$V(x)$ is a column-major 3D tensor whose elements are 3D tiles. Each 3D tile, $V(x)(ii, jj, kk)$, is stored as a row-major 2D tensor whose elements are 1D arrays of floating point values, oriented according the z-axis. The tile object implements an `operator() (const int i, const int j)` for extracting the z-vector indexed by i and j. The velocities are block-wise computed, where each block is a z-vector of coordinates (i, j) within the considered tile. Prior computations, each thread will load a z-vector of their respective tiles into the L1 cache of the core on which it is bound. Hence, with an appropriate tile size, for all (i_0, j_0) and (i_1,j_1), the z-vectors $V(x)(ii, jj, kk)(i_0, j_0)$ and $V(x)(ii+1, jj, kk)(i_1, j_1)$ will never be on the same cache line. Consequently, the proposed approach minimizes the probability of *false-sharing* during execution. Moreover, by appropriately adjusting the tile size, all its data can fit into cache, hence increasing the arithmetic intensity of the computation, required for a better vectorization efficiency.

Vectorization efficiency: computing V on a single tile involves a series of partial derivatives of S along the three spatial dimensions. Along both x and y dimensions, the computations are similar across all cells for a fixed (i, j), and can thus be performed in parallel. To leverage this fine-grained parallelism, the data layout has been designed so that each tile is a 2D grid of 1D vectors along the z-axis. For instance, $V(x)(ii, jj, kk)(i, j)$ is a 1D vector, eventually padded with additional cells to match the SIMD width on the target architecture. This strategy allows us for explicitly computing the derivatives using SIMD instructions. The same analysis holds for the computation of S.

4 A Hierarchical Implementation Tailored for Modern Architectures

4.1 Implementation on top of PaRSEC

Emergence of task scheduling engines: The past years have witnessed the emergence of generic task-based runtime systems [19–21]. These systems were introduced to cope with the application development issues arising since the advent of massively parallel and heterogeneous computing. Such a system offers a unified view of the underlying hardware and let the developer focus on the algorithm, described as Directed Acyclic Graph (DAG) of tasks. The runtime system will then manage all data transfers and synchronizations between computing devices (CPUs/GPUs/MICs) and the scheduling of tasks among available computing resources. Hence, these frameworks allow for the separation of major concerns in HPC: design of the algorithm, creating a data distribution and developing computational kernels.

While the general principle of the runtime systems is similar for all, two major tendencies exist and differ according to the DAG of tasks construction: Parametrized Task Graph (PTG) [22] and Dynamic Task Graph (DTG) models [23]. In the PTG approach, the DAG is constructed as a problem size independent symbolic representation of the algorithm and can thus be generated

Listing 1.1: ComputeVelocity task in the JDF language.

```
1  ComputeVelocity(d, t, ii, jj, kk)
2  d  = 0 .. dim-1
3  t  = 2 .. nt-2 .. 2
4  /* ii, jj, kk from 0 to nxx-1, nyy-1, nzz-1 */
5
6  : ddesc(ii, jj, kk)
7
8  CTL SxxH<- ( d==X          && t > 2) ? SxxH UpdateStress(XX, t-1, ii, jj, kk)
9  CTL SxyH<- ((d==X || d==Y) && t > 2) ? SxyH UpdateStress(XY, t-1, ii, jj, kk)
10 CTL SxzH<- ((d==X || d==Z) && t > 2) ? SxzH UpdateStress(XZ, t-1, ii, jj, kk)
11
12 BODY
13 {
14   computeVelocity(d, t, ii, jj, kk);
15 }
16 END
```

at compile time. Hence, the instantiation of new tasks is performed during execution through a closed formula depending on the task parameters. Such an approach is implemented by the PaRSEC [21] framework, offering a specific annotation-based language, the Job Data Flow (JDF), for describing the DAG of tasks according to their INPUT and OUTPUT data. Conversely, with the DTG approach, the DAG of tasks is fully constructed and kept in memory, which the runtime system is going to explore during execution for discovering and scheduling ready tasks. Thus, the DAG memory occupation grows with the problem size. Nevertheless, DTG frameworks (e.g. StarSS [19] and StarPU [20]) usually allow to customize a window of visible tasks for avoiding the full generation of the DAG of tasks at runtime. An appropriate window size can therefore help reducing the memory footprint with a minimal performance penalty. Given the regular pattern of the spatial mesh we considered in this study, we choose to implement our data flow on top of the PTG-based framework PaRSEC.

Implementation: Building an application on top of PaRSEC requires to define the algorithm data-flow using the JDF language and a data distribution that will be used by the runtime system for tasks placement on the target architecture. The Listing 1.1 shows a simplified version of the JDF implementation of the ComputeVelocity task. There are four main parts in the task description.

1. The execution space (Line 2 to Line 4) is defined by a valid range, for each of the task parameters, determining the total number of similar tasks.
2. The parallel partitioning (Line 6) is a symbolic reference to a data element that is going to be used by the runtime system to execute the task according to the owner compute rule. Basically, the task will be scheduled on the node where the data element is located.
3. Task data-flow (Line 8 to Line 10) defines the input and output dependencies of the task, eventually conditioned by a C-style ternary operator. Here, the keyword CTL (Control flow) is used as a counter by the runtime system (not a real data transfer). For instance, computing V_x requires the reception

Listing 1.2: UpdateStress task in the JDF language.

```
1   UpdateStress(s, t, ii, jj, kk)
2   s = 0 .. nsc-1
3   t  = 1 .. nt-3 .. 2
4   /* ii, jj, kk from 0 to nxx-1, nyy-1, nzz-1 */
5
6   : ddesc(ii, jj, kk)
7
8   READ SLeft  <- (ii>0 && t>1)     ? SRight ExtractStressH(s, t, ii-1, jj, kk)
9   READ SRight <- (ii<nxx-1 && t>1) ? SLeft  ExtractStressH(s, t, ii+1, jj, kk)
10
11  CTL SxxH    -> (s==XX && t>1)    ? SxxH ComputeVelocity(X, t+1, ii, jj, kk)
12  CTL SxyH    -> (s==XY && t>1)    ? SxyH ComputeVelocity(X, t+1, ii, jj, kk)
13  CTL SxzH    -> (s==XZ && t>1)    ? SxzH ComputeVelocity(X, t+1, ii, jj, kk)
14
15  BODY
16  {
17    updateStress(LEFT,  s, t, ii, jj, kk, SLeft);
18    updateStress(RIGHT, s, t, ii, jj, kk, SRight);
19  }
20  END
```

of three controls: Line 8, Line 9 and Line 10, notifying the completion of UpdateStress task on respectively xx, xy and xz components of the stress field. In Listing 1.2, we can see the matching control flows sent by UpdateStress task at Line 11, Line 12 and Line 13. Also, UpdateStress has an input data dependency as indicated on Line 11. This line specifies that UpdateStress will create a temporary read-only (READ keyword) buffer SLeft where an output data SRight sent by ExtractStressH will be stored.

4. The task body between BODY and END keywords contains the code executed by the task. Here, the body is given at Line 14 as a function call to our implementation of the velocity computation on a single tile.

This JDF is complemented with a data distribution implementation, through two PaRSEC provided functions, evaluated by all processes: `data_of` and `rank_of`. The former returns the pointer to the actual data described by `ddesc(ii, jj, kk)` and the latter returns the rank of the MPI process holding that data. The combination of the JDF and the data distribution will be used by PaRSEC to schedule and execute tasks, according to the provided computational kernels.

4.2 Building Generic Optimized Computational Kernels

To build the kernels, we considered three metrics: expressivity, performance and performance portability across various architectures. In the following, we demonstrate how our implementation managed to maximize these metrics.

Expressivity: We adopted the C++ language that allows for building complex and meaningful expressions, close to the mathematical formulations used in an algorithm. Let us consider the V_x computation as given in Listing 1.3. The code shows a tile traversal in the (x, y) plane. For each position (i, j), velocities of all

Listing 1.3: Computation of V_x on a single tile.

```
1  for (int i=iStart; i<iEnd; i++){
2    for (int j=jStart; j<jEnd; j++){
3      vX(i,j) +=(fdo_.apply<SWS::X>(sigmaXX, i, j)
4              +  fdo_.apply<SWS::Y>(sigmaXY, i, j)
5              +  fdo_.apply<SWS::Z>(sigmaXZ, i, j))*dt*bx(i,j);
6    }
7  }
```

	$T_{\max}$	dt	N_t	l_x	l_y	l_z	ds	n_x	n_y	n_z
TestA	1.6	0.008	200	20	20	10	0.1	200	200	100
TestB	20	0.2	2000	650	1000	50	0.5	1300	2000	100

Table 2: Characteristics of TestA and TestB benchmarks.

cells along the z axis are computed at Line 3, according to block-wise evaluations. `fdo_` is an object of type `CentralFDOperator`, a class implementing the $4th$ order central finite-differences scheme through the member function `apply`, templated with the derived direction. We can notice the similarity between expression on Line 3 with the numerical formulation of V_x as shown in (1).

Performance and Performance Portability: Using Expression Templates to build our containers on top of the generic C++ Eigen library [24] avoids the creation of temporary 1D vectors for each each call to the `apply()` method on Line 3 of Listing 1.3. In addition, Eigen supports explicit vectorization with various SIMD extensions, including SSE2, AVX2, AVX-512, and ARM NEON allowing the application to be portable across a large number of architectures.

5 Experiments

We conducted strong scalability studies on the Frioul supercomputer from GENCI/CINES[4], based on Intel KNL 7250, comprising 68 cores running at 1.4 GHz. There is 16 GB of MCDRAM on-chip memory per node and 192 GB of DDR4 off-chip memory. The computing nodes are interconnected through an InfiniBand EDR fabric, providing a theoretical bandwidth of 100 Gb/s. All the experiments have been carried out using two test cases: TestA and TestB (see Table 2). The former is a small test case and the latter, representing a real earthquake, is larger. In the following, both SeWaS and Ondes3D applications are compiled using Intel compiler version 18.0.1 and Intel MPI version 5.1.3.

5.1 Tuning Single Node Performances

As mentioned in Section 3, the performances of the application is strongly dependent of the tile size. To evaluate the best size, we considered the TestA

[4] `https://www.cines.fr/le-supercalculateur-frioul/`

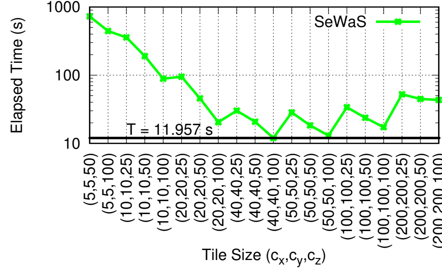
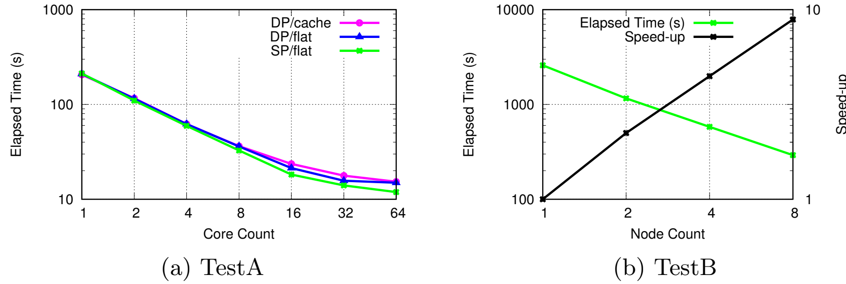


Fig. 3: Evaluation of the best tile size running TestA in single precision.



(a) TestA

(b) TestB

Fig. 4: SeWaS strong scaling. The left curve shows the single-node performance using TestA. It compares the performances of single and double precisions for both KNL configurations (cache and flat modes). The right curve shows the multi-node performance using TestB where all nodes are configured on flat mode.

benchmark and compared the computation time of SeWaS for different configurations of (c_x, c_y, c_z) on a single 64-cores node. The results are presented in Figure 3. For the considered test case, the absolute discrepancy compared to the double precision results is an order of 10^{-4} which is acceptable for the purposes of our experiments.

We found that the best computation time (11.957 s) is obtained with a tile of size $(40, 40, 100)$. For this size, the total number of tasks is 67500, that is 1054 tasks per thread. We can also notice that for a fixed c_x and c_y, the best computation time is obtained when using c_z is 100. This result is justified by the fact the computations are vectorized along the z axis, and thus the performances tend to be better when c_z is large enough to fit the SIMD units. In the following, all the presented results were obtained using the determined best tile size.

Figure 4a presents a single-node strong scalability study of SeWaS using TestA benchmark. It shows that the run with flat mode is slightly faster than with cache mode.

In double precision, SeWaS computation time on 64 cores is 14.9 s, whereas Ondes3D takes 56.9 s. This difference is due to the explicit vectorization used

$T_{\max}$ (s)	1	2	4	8	16	20
N_t	100	200	400	800	1600	2000
Computation Time (s)	108.3	117.6	136.7	175.4	252.6	291.5

Table 3: Illustration of time-steps overlapping.

in SeWaS and a better data locality. Even if Ondes3D is a more generalistic application and does implement other features that are not present in SeWaS, such as absorbing boundary condition, we expect that this comparison will give to the reader an order of magnitude of how SeWaS compares to Ondes3D. In the following, all results are obtained in single precision and using the flat mode.

5.2 Distributed Memory Scaling

We consider the TestB benchmark. This test case contains 1300 cells along the x dimension. As it is not divisible by 40, we will be using the $(50, 50, 100)$ tile size whose performance are very close.

Strong scaling: The Figure 4b presents SeWaS strong scaling using TestB benchmark on 8 nodes. The benchmark runs in 2589.7 s on a single node, and 291.8 s using 8 nodes, corresponding to a speed-up of 8.8. The super linear scalability observed is due to the fact the computation on a single node requires around 40 GB of memory which is larger than the size of the on-chip MCDRAM. Indeed, on a single node, 24 GB of data will be allocated in DDR4. Starting from 4 nodes, all data can fit in the MCDRAM, and thus the performances are improved.

Impact of time-steps overlapping: We conducted an experiment to study the behavior of computations overlap for successive time-steps in SeWaS. We measured the computation time for several values of $T_{\max}$ from 2 s to 20 s. The results are presented in Table 3. We observe that the computation time increases with $T_{\max}$, following a linear trend experimentally determined as:

$$\text{Computation Time} \approx T_{\text{init}} + 9.7 * T_{\max},$$

where $T_{\text{init}} \approx 98.6$ s is the time spent to initialize the computations. Let us first consider the core simulation time, that is the computation time without the initialization. We notice that the ratio of the core simulation time between $T_{\max} = 20$ s and $T_{\max} = 2$ s is 7.8 representing an overlap ratio of 22%. This is a remarkable result as it shows a high overlapping rate for the computations of different time-steps. A preliminary experiment showed that the initialization can be fully parallelized allowing to mitigate its impact on the computation time.

6 Conclusion

In this paper, we presented the design and implementation of SeWaS, a linear seismic wave propagation code, adapted for modern computing platforms.

We studied the main challenges related to the development of efficient and scalable computation code on these platforms. A fully task-based model has been designed and its implementation combines the state-of-the-art frameworks and libraries PaRSEC and Eigen. Performance studies conducted on a cluster of Intel KNL processors showed that the application exhibits a good strong scalability up to 8 nodes. The proposed approach demonstrated a clear path toward code modernization required to take advantage of computing power brought by current and coming Exascale systems. In the future, we will extend our computational kernels for GPUs to cope with highly heterogeneous systems.

Data Availability Statement and Acknowledgments

The datasets generated during and/or analyzed during the current study are available in the Figshare repository: `https://doi.org/10.6084/m9.figshare.6387743`. We would like to thank GENCI and CINES for providing us computing facilities to perform our experiments.

References

1. Ross, P.E.: Why cpu frequency stalled. IEEE Spectrum **45**(4) (2008)
2. Moustafa, S., Faverge, M., Plagne, L., Ramet, P.: 3D Cartesian Transport Sweep for Massively Parallel Architectures with PaRSEC. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE (2015) 581–590
3. Taylor, R.A., Jeong, J., White, M., Arnold, J.G.: Code modernization and modularization of APEX and SWAT watershed simulation models. International Journal of Agricultural and Biological Engineering **8**(3) (2015)
4. Jundt, A., Tiwari, A., Ward Jr, W.A., Campbell, R., Carrington, L.: Optimizing codes on the Xeon Phi: a case-study with LAMMPS. In: Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure, ACM (2015) 28
5. McKee, S.A.: Reflections on the memory wall. In: Proceedings of the 1st conference on Computing frontiers, ACM (2004) 162
6. Kirschenmann, W., Plagne, L., Vialle, S.: Multi-target C++ implementation of parallel skeletons. In: Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing, ACM (2009) 7
7. Furumura, T., Chen, L.: Large scale parallel simulation and visualization of 3D seismic wavefield using the Earth Simulator. Computer Modeling in Engineering and Sciences **6** (2004) 153–168
8. Aochi, H., Ulrich, T., Ducellier, A., Dupros, F., Michea, D.: Finite difference simulations of seismic wave propagation for understanding earthquake physics and predicting ground motions: Advances and challenges. In: Journal of Physics: Conference Series. Volume 454., IOP Publishing (2013) 012010
9. Virieux, J., Madariaga, R.: Dynamic faulting studied by a finite difference method. Bulletin of the Seismological Society of America **72**(2) (1982) 345–369
10. Graves, R.W.: Simulating seismic wave propagation in 3D elastic media using staggered-grid finite differences. Bulletin of the Seismological Society of America **86**(4) (1996) 1091–1106

11. Kristek, J., Moczo, P.: Seismic-wave propagation in viscoelastic media with material discontinuities: A 3D fourth-order staggered-grid finite-difference modeling. Bulletin of the Seismological Society of America **93**(5) (2003) 2273–2280
12. Roten, D., Cui, Y., Olsen, K.B., Day, S.M., Withers, K., Savran, W.H., Wang, P., Mu, D.: High-frequency nonlinear earthquake simulations on petascale heterogeneous supercomputers. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016. (2016) 957–968
13. Breuer, A., Heinecke, A., Bader, M.: Petascale Local Time Stepping for the ADER-DG Finite Element Method. In: 2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016. (2016) 854–863
14. Göddeke, D., Komatitsch, D., Geveler, M., Ribbrock, D., Rajovic, N., Puzovic, N., Ramírez, A.: Energy efficiency vs. performance of the numerical solution of PDEs: An application study on a low-power ARM-based cluster. J. Comput. Physics **237** (2013) 132–150
15. Castro, M., Francesquini, E., Dupros, F., Aochi, H., Navaux, P.O.A., Méhaut, J.: Seismic wave propagation simulations on low-power and performance-centric manycores. Parallel Computing **54** (2016) 108–120
16. Sornet, G., Dupros, F., Jubertie, S.: A Multi-level Optimization Strategy to Improve the Performance of Stencil Computation. Procedia Computer Science **108** (2017) 1083–1092
17. Moustafa, S., Kirschenmann, W., Dupros, F., Aochi, H.: Code and input data for SeWaS: Seismic Wave Simulator: Euro–par 2018 artifact. figshare. Code. `https://doi.org/10.6084/m9.figshare.6387743` (2018)
18. Dupros, F., Aochi, H., Ducellier, A., Komatitsch, D., Roman, J.: Exploiting intensive multithreading for the efficient simulation of 3D seismic wave propagation. In: Computational Science and Engineering, 2008. CSE'08. 11th IEEE International Conference on, IEEE (2008) 253–260
19. Planas, J., Badia, R.M., Ayguadé, E., Labarta, J.: Hierarchical task-based programming with StarSs. The International Journal of High Performance Computing Applications **23**(3) (2009) 284–299
20. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**(2) (2011) 187–198
21. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.: DAGuE: A generic distributed DAG engine for high performance computing. Parallel Computing **38**(1) (2012) 37–51
22. Danalis, A., Bosilca, G., Bouteiller, A., Herault, T., Dongarra, J.: PTG: an abstraction for unhindered parallelism. In: Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, IEEE Press (2014) 21–30
23. Advea, V., Sakellariou, R.: Compiler synthesis of task graphs for parallel program performance prediction. In: International Workshop on Languages and Compilers for Parallel Computing, Springer (2000) 208–226
24. Guennebaud, G., Jacob, B., et al.: Eigen v3. http://eigen.tuxfamily.org (2010)