Parallel and distributed task-based Kirchhoff seismic pre-stack depth migration application

Jerome Gurhem, Henri Calandra, Serge G Petiton

Parallel and Distributed Task-Based Kirchhoff Seismic Pre-Stack Depth Migration Application

Jérôme Gurhem
UMR 9189 - CRIStAL, Univ. Lille, CNRS
Lille, France
USR 3441 - Maison de la Simulation, CNRS
Saclay, France
0000−0002−0741−9766

Henri Calandra
Total SA
Pau, France
henri.calandra@total.com

Serge G. Petiton
UMR 9189 - CRIStAL, Univ. Lille, CNRS
Lille, France
USR 3441 - Maison de la Simulation, CNRS
Saclay, France
0000−0002−8423−3682

Abstract—Since the middle of the 1990s, message passing libraries are the most used technology to implement parallel and distributed scientific applications. However, they may not be a solution efficient enough on exascale machines since scalability issues will appear due to the increase in computing resources. Task-based programming models can be used to avoid collective communications like reductions, broadcast, or gather by transforming them into multiple operations on tasks. Then, these operations can be scheduled by the programming scheduler to place the data and computations in a way that optimizes and reduces the data communications. These properties could help to solve some MPI and exascale computing challenges.

The oil and gas applications could also benefit from task-based programming properties. We developed a simplified version of the Kirchhoff seismic pre-stack depth migration, a subsurface exploration application, to experiment with HPX, a task-based programming model as well and MPI and MPI+OpenMP. Then, we perform strong scaling and weak scaling experiments on Pangea, Total supercomputer. We also study the variation of the number of OpenMP threads per MPI process. We show that the current task-based programming model schedulers lack the capability to completely manage the memory used and are not efficient enough to reduce the data migrations.

Index Terms—Kirchhoff Seismic Pre-Stack Depth Migration, Task-Based Programming, Parallel and Distributed Application

Introduction

On the current post petascale supercomputers, MPI is the most common library used to implement parallel and distributed applications. MPI is used to send data between supercomputer nodes and is often associated with OpenMP [1] or CUDA [2]. OpenMP can be used to implement multi-threaded code that will be executed on the CPU. On the other hand, CUDA can be used to address Nvidia accelerators programming. However, MPI+X may not be a solution efficient enough on exascale machines, especially in terms of fault tolerance, check-pointing [3] and collective communications scalability. A task-based approach can help in managing fault tolerance and check-pointing since the tasks could be restarted on another location and data from tasks saved at any moment. Task-based programming models can implement fault tolerance by restarting the failed tasks on other computing resources and even stop executing tasks on processors with a high fault rate. Check-pointing could be implemented by storing the state of the output of the task in a way that allows to restart the application at the last check-point and reconstruct the lost data. Moreover, task-based programming models can also be used to avoid collective communications like reductions, broadcast, or gather by transforming them into multiple operations on tasks. Then, these operations can be scheduled by the programming scheduler to place the data and computations in a way that optimizes and reduce the data communications. Therefore, with the merging of parallel and distributed programming, graphs of tasks and efficient schedulers are a very interesting way to improve performances while reducing communications.

The oil and gas applications [4] could benefit from task-based programming properties [5]. It has also been shown that task-based programming models can obtain better performances than regular applications for several dense linear algebra [6]. Thus, in this paper, we try to explore the possibilities of task-based programming models with scientific applications. For instance, subsurface exploration needs a large number of computing resources to process the large volume of data acquired during the subsurface surveys. Then, these data have to be efficiently scheduled through the application from their reading from the file system to their process in several parts of the application. We implemented a simplified 2D version of the Kirchhoff seismic pre-stack depth migration to implement and perform experiments with task-based programming models.

In the first section, we introduce task-based programming paradigms. Then, in the second section, we present the Kirchhoff seismic pre-stack depth migration and its related algorithms. Furthermore, in the third section, we give details about our implementation of the application. Afterward, we show the results of our experiments. Finally, in the last section, we discuss task-based programming models as an alternative to MPI for the Kirchhoff seismic pre-stack depth migration.

I. Task-Based Programming

Collective communications like reductions, gathers, and broadcasts are very expensive due to the high number of resources partaking in the operation and the cost of sending information to distant resources on the network connecting the nodes. A task-based approach can help in managing fault tolerance and check-pointing since the tasks could be restarted on another location and data from tasks saved at any moment. Tasks allow separating the expression of the parallelism from its parallel implementation by letting the developer express the tasks and their dependencies while the runtime of the programming models tries to run as many tasks as possible at the same time, respects the dependencies, and tries to obtain the best performances possible. This means that application experts can express algorithms through graphs of tasks without being required to understand the hardware in detail.

Furthermore, the task-based approach can help to eliminate large-scale collective communications by encapsulating them inside tasks. Then, these tasks can run on a subset of the resources allocated to the application and execute collective communications on a smaller scale. The graph of tasks can be efficiently scheduled so that the execution of tasks optimizes data migrations, IOs tasks, and data check-pointing. In task-based programming models, data can only be exchanged between tasks as their input and output parameters as opposed to exchanging data during the execution of the task. Therefore the algorithms using collective communications have to be redesigned to avoid them or rewrite them as task operations.

A task can be defined as a set of encapsulated operations asynchronously executed. It can range from a few computations to a fully distributed application. They can only communicate with their input and output parameters so that the programming model runtime can efficiently manage data migrations and schedule tasks accordingly. Data and/or control dependencies have to be provided by the user. These dependencies are enforced during the execution of the task-based application. The runtime tries to execute as many tasks as possible.

We choose to use HPX for our experiments due to its properties. HPX is an extension of the C++ standard library to the distributed case. It provides an interface for task-based programming through C++ concurrency facilities such as dataflows and futures. The tasks are executed as lightweight threads. These tasks are relatively fine grain. HPX has interesting results on a low number of nodes. It ran faster than MPI to perform a block-based LU factorization as shown in [6].

Task-based programming models and their properties [7] were explored previously. They express the key properties of task-based programming in different ways. In this paper, we explore the abilities of HPX with a subsurface exploration application.

II. Kirchhoff Seismic Pre-Stack Depth Migration

In this section, the algorithms for the Kirchhoff seismic pre-stack depth migration are introduced. Then, the parallelism approaches of the method are discussed Finally, a task-based algorithm for the Kirchhoff seismic pre-stack depth migration is presented.

A. General method

Seismic migration is a technique used to visualize the underground. Data are acquired at the surface during an acquisition campaign. Data are processed to geometrically re-locate seismic events either in space or in time. They are re-located to the location the event occurred in the subsurface rather than the location where it was recorded at the surface. Migration moves dipping reflectors to their true subsurface positions and collapses diffractions, resulting in a migrated image that typically has an increased spatial resolution and resolves areas of complex geology much better than non-migrated images. A form of migration is one of the standard data processing techniques for reflection-based geophysical methods (seismic reflection and ground-penetrating radar).

The Kirchhoff migration [8] [9] is a depth migration. It is applied to seismic data in depth (regular Cartesian) coordinates, which must be calculated from seismic data in time coordinates. This method does therefore require a velocity model, making it resource-intensive because building a seismic velocity model is a long and iterative process. The significant advantage of this migration method is that it can be successfully used in areas with lateral velocity variations, which tend to be the areas that are most interesting to petroleum geologists.

The velocity model describes the propagation speed of the waves into the different layers of the underground. Geophysicists create a model and want to verify its correctness. Their goal is to find the best model that explains the data. They use the Kirchhoff migration that determines where are the limits between layers.

The Green functions represent the response and the behavior of the wave when the source is a Dirac impulse. They allow solving the wave equation using an integral formula in function of the source of the wave. The behavior of the wave in the ground depends on the velocity model. Usually, Green functions are precomputed and stored on a disk for a 2D grid on the surface and a 3D grid underground. During the building of the model and the migration, Green functions are retrieved from the disk. To reduce the IOs with the file system, the Green functions are computed on a coarser grid than the one considered for the image migration.

The Kirchhoff migration produces a 3D image of the subsurface by retrieving the position of the reflection points to show the different layers of the ground. To do so, we find the time a wave needs to travel from a source to a point (x, y, z) in the image, and travel back from this point to the receiver is necessary. The Green functions in the

fine grid give this time. For a given source and receiver, if the time a wave needs to travel through the point (x, y, z) and the time to the peak of the trace match then (x, y, z) can be a candidate to a reflection point. Moreover, we know the Green functions for a coarse grain of the image, so it is possible to know if the points in the sub-domain of the coarse grid will match the time from the trace. If the time matches, the points will be studied at a finer grain. Figure 1 shows the two grid levels; the corners of the red blocks represent the coarse grid while the inside contains the finer grid. Otherwise, the points of the part of the coarse grid will not be candidates to a reflection point and those points will be ignored for the selected trace.
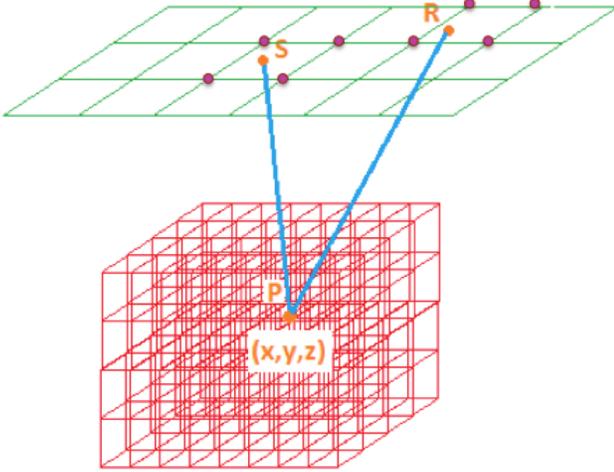

Fig. 1. Fine and Coarse grids for subsurface image

At the finer grain level, the Green functions are interpolated or extrapolated for each point in the block. This gives the travel time $T(x, y, z)$ between the source and the receiver passing by the point (x, y, z). Then, only the points matching the time of the trace are kept. Those points are likely true reflection points. An arbitrary value (the aperture) $A_{s,r}(x, y, z)$ [10] that favors points with less awkward reflection angles, and positions that are more likely to match the true reflection point, can be calculated. This value depends on the coordinates of the receiver 'r', the coordinates of the source 's', and the point (x, y, z). The value $A_{s,r}(x, y, z) t_{s,r}(x, y, z)$ expresses the intensity of the contribution of the trace for the point (x, y, z).

The image at a pixel is generated by summing the contribution of all the traces that can be a true reflection point :

$$I(x, y, z) = \sum_{t_{s,r} \in T} A_{s,r}(x, y, z) t_{s,r}(\tau_s(x, y, z) + \tau_r(x, y, z))$$

(1)

$I(x, y, z)$ is the pixel (x, y, z) of the image. $t_{s,r}$ is the trace which has s as source and r as receiver. T is the ensemble of traces. $A_{s,r}$ is the amplitude associated to s

and r. $\tau_j(x, y, z)$ is the time need for a wave to travel from j (a source or a receiver) to (x, y, z).

```
foreach trace t do
    r = extractReveiver(t)
    s = extractSource(t)
    Gs = loadPrecomputedGreenFunc(s)
    Gr = loadPrecomputedGreenFunc(r)
    foreach (x, y, z) in the coarse grid do
        /* contains values for (x, y, z), (x+1, y, z),
           (x, y+1, z), (x, y, z+1), (x+1, y+1, z), (x,
           y+1, z+1), (x+1, y, z+1) and (x+1, y+1,
           z+1) */
        TS = computeTravelTime(Gs, x, y, z)
        TR = computeTravelTime(Gr, x, y, z)
        A = computeAmplitude(s, r, x, y, z)
        foreach (i, j, k) in the fine grid do
            Ts = interpolate(i, j, k, x, y, z, TS)
            Tr = interpolate(i, j, k, x, y, z, TR)
            Ai = interpolate(i, j, k, x, y, z, A)
            Img(i, j, k) += Ai × ts,r(Ts + Tr)
```

Algorithm 1: General Kirchhoff seismic pre-stack depth migration

Algorithm 1 shows a possible high-level implementation of the Kirchhoff seismic pre-stack depth migration as presented in this section. In this algorithm, the functions extractReveiver and extractSource are used to extract the position of the receiver and the source of the trace data. Then, these positions are used to load the precomputed Green functions from disk through the functions loadPrecomputedGreenFunc. The loaded Green functions are used to compute, in the computeTravelTime function, the travel time that a sound wave will need to travel from the source to the receiver passing by a position in the coarse grid. The interpolate function uses these travel times to interpolate them for positions in the fine grid so that the image at this position can be computed.

B. Parallelism

In the Kirchhoff Migration, only the output image is modified. The traces and the Green functions are accessed only for reading. Therefore, the update of the image by the contribution of a trace conditions the parallelism available. Moreover, the traces are independent and the update of a point on the image depends only on the contribution of the trace.

It means that all the points can be updated at the same time by a trace. It also means that there are concurrency problems if a point is updated by two traces. Since the update is a sum, it is worth considering creating images for different sets of traces independently, then reduce (sum) the different images into a final image.

From a computational point of view, traces can be treated in any order and points can be computed independently. Several traces can be treated at the same

time with reductions on the output images. So there is a lot of available computational parallelism but there is also data to transfer between compute units and from the file system. Moreover, creating several images to reduce them later also produces communications to reunite them into one. In the case where the Green functions are precomputed, loading them into the memory can be an issue. They take a lot of space on disk so it is expensive to load them several times. Thus, scheduling those data movements can be a good alternative to find an efficient way to manage the data without having to communicate too much. To better express the dependencies, the method is described as a graph of tasks.

C. Task-Based Method

In the Kirchhoff Migration, the only data modified is the output image and the inputs are only accessed for reading. Therefore, all the parallelism of the method is related to the accesses to the image and the IOs with the file system to fetch the traces and Green functions. There are two main sources of parallelism: splitting the image to work on multiple sub-images and splitting the trace set to work on them in parallel.

The latter method creates several images that have to be combined. It is only a sum but it requires migrating images between the computing resources. On the other hand, the data from the trace will have to be duplicated for each sub-image but only the Green functions for the sub-part of the image can be loaded.

```
foreach trace t do
    r = extractR(t)
    s = extractS(t)
    foreach sub-block b from the coarse grid do
        Gs = loadPrecomputedGreenFunc(s, b)
        Gr = loadPrecomputedGreenFunc(r, b)
        migrateSubBlock(b, t, Gs, Gr)
```

Algorithm 2: Task-Based Kirchhoff seismic pre-stack depth migration

We choose to implement the method with distributed images. Algorithm 2 shows the available high-level parallelism of the method. This is also a task-based algorithm in which the tasks consist of migrating traces into a sub-image. The task corresponds to the interpolation of the travel time and the sum with the image introduced in Algorithm 1. This algorithm is implemented with MPI, MPI+OpenMP [1] and HPX [11], a task-based programming model in which dependencies between tasks are data-oriented.

III. Application Description

For this implementation, we simplified the Kirchhoff seismic pre-stack depth migration by considering the absorption to be equal to 1 and by only studying the 2D case. This implementation is a very basic and simplified version of the Kirchhoff seismic pre-stack depth migration which keeps a similarly high-level algorithm while simplifying the computations needed to perform the migration on the image points.

A. C Kernel Description

The kernel performing the main operations for this method is implemented in C and provides functions to manage traces, Green functions, and images. There are three kinds of functions implemented in this kernel. The first type is IOs to read and write traces, images, and Green functions from files. The second type of function is used to associate propagation time precomputed with Green functions to the trace that is used to create the image. Then, the last type is the actual migration.

The migration function expects a fine grain image to update and a trace with the propagation times to make the round trip from the source of the trace to the receiver passing by every point of the large grain image. It performs an interpolation to compute the propagation time needed to travel from the source to the receiver for each point of the fine-grain image by using the propagation times shipped with the trace. So, with the knowledge of the time, it would take to make the round trip to the considered image point, we scale it to the trace time scale to retrieve the amplitude of the sound wave at this time in the trace. Then, the amplitude is added to the value at the corresponding point in the fine-grain image. This amplitude could be scaled with the absorption at this point but we set the absorption to 1 for this implementation so this is not necessary here. The migration function has OpenMP directives to parallelize the work on the image. They can be activated by compiling the library with OpenMP.

For our experiments, we use a constant velocity model to generate the initial propagation times. Therefore, we implemented a function to interpolate propagation times on a large grain grid from the constant velocity model. A function to generate simple traces has also been implemented to use in our experiments.

Figure 2 gives, on the top, an example of a generated trace used as input for our applications. In the middle, this figure provides a plot of the propagation times to make the round trip from the source point located at (500,0), each point of the figure, and the receiver located at (500, 0) in a uniform medium. On the bottom, the plot shows the migrated image produced by the Kirchhoff migration with the trace on the top of the figure taken at several different source locations (the source and the receiver are located at the same place).

B. Distributed and Parallel Implementations

Distributed and parallel applications were implemented on top of the C kernel in MPI and HPX.

The MPI application uses the C functions to generate the traces and propagation time associated with the

The HPX application is based on the same principles. The image is split and managed by tasks. A task calling the migration available in the kernel has been implemented as well as tasks to generate the traces and the propagation times. There are also tasks to associate the trace with the appropriate propagation times. In Figure 3, the first HPX function requires the data needed for the migration and calls the C kernel to perform the migration. This function defines an HPX task. Then, this task is used in the following function to express data dependencies between the tasks with HPX dataflow function. This function iterates over all the traces and sub-images to launch the tasks to perform the migration on the input traces. Afterward, the tasks can be efficiently executed by the HPX scheduler to obtain the migrated image.

```
static hpx_::client::Image hpx_::detail::
    migrate_trace(hpx_::client::Image const& I
    , hpx_::client::Trace const& T) {
  using hpx::dataflow;
  using hpx::util::unwrapping;

  hpx::shared_future<hpx_::detail::TraceBase>
      T_data = T.get_data();
  hpx::shared_future<hpx_::detail::ImageBase>
      I_data = I.get_data();
  return dataflow(hpx::launch::async,
    unwrapping([I](hpx_::detail::ImageBase
        const& i_, hpx_::detail::TraceBase
        const& t_) -> hpx_::client::Image {
      hpx_::detail::ImageBase i(i_);
      i.migrate_trace(t_);
      return hpx_::client::Image(I.get_id(), i
        );
    }),
    I_data, T_data);
}

HPX_PLAIN_ACTION(hpx_::detail::migrate_trace,
    migrate_trace_action);

void hpx_::Image::kirchhoff_migration() {
  migrate_trace_action act_m;
  using hpx::dataflow;
  std::size_t incr = 0;
  for (std::size_t t = 0; t < trace.get_nt();
    ++t) {
    for (int i = 0; i < images.size(); i++) {
      using hpx::util::placeholders::_1;
      using hpx::util::placeholders::_2;
      auto Op = hpx::util::bind(act_m,
          localities[i * localities.size() /
          images.size()], _1, _2);
      images[i] = dataflow(hpx::launch::async,
          Op, images[i], trace.get_trace(incr
          ));
      incr++;
    }
  }
}
```

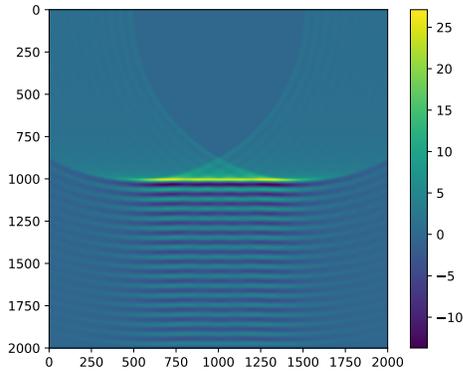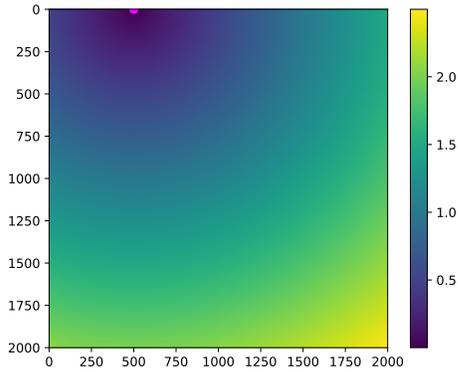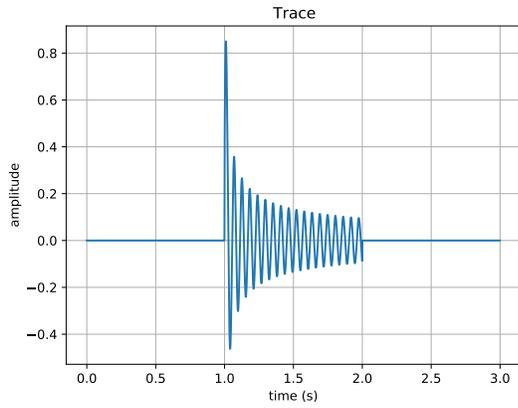Fig. 3. HPX Kirchhoff migration task implementation and execution

Fig. 2. Kirchhoff Migration - generated sample trace (top), wave propagation times for a source and receiver located at (500, 0) (middle), migrated image (bottom)

distributed processes. The image is distributed across the processes allocated to the application. The traces are duplicated on each process since they are accessed depending on the travel time extracted from the propagation time shipped with the trace. The propagation times are generated with the trace and only the part that is used in the migration is selected. Then, the migration can be executed on the local image with the local trace and propagation times.

These applications support the generation and processing of several traces. They were designed to perform scaling experiments on the migration with a distributed image.

IV. Numerical Experiments

In this section, we perform numerical experiments on the Kirchhoff seismic pre-stack depth migration implemented in MPI, MPI+OpenMP, and HPX on Pangea II. We perform strong scaling and weak scaling experiments as well as we study the influence of the number of OpenMP threads on our kernel.

A. Pangea II

Pangea II is the supercomputer on which the experiments have been performed. It is owned by Total and located in Pau, France. This supercomputer is composed of nodes built with 2 Xeon E5-2680v3 12C 2.5GHz processors for a total of 220 800 cores. The nodes are connected with Infiniband FDR interconnects. It performs at 5.283 PFlop/s for Linpack and 162.692 TFlop/s for HPCG.

B. Strong Scaling

Strong scaling experiments were performed on a 15000 × 15000 points image. We executed our Kirchhoff seismic pre-stack depth migration MPI, MPI+OpenMP and HPX applications on 10 generated traces with propagation times. We study the performance improvement of the migration while the number of nodes (cores) increases and the size of the image is kept constant.

Figure 4 shows the results of our strong scaling experiments on the Kirchhoff seismic pre-stack depth migration for an image of the size of 15000 × 15000 and 10 traces. HPX execution times are increasing with the increase of computing resources which should not be the case.

HPX may induce image migration between nodes that are not necessary. HPX assigns data to a locality that represents a physical node. A trace may be assigned to a different location than one of the images on which the trace will be migrated. At this moment, either the image or the trace has to be migrated to compute the new image. The image is larger than the trace so if the image is transferred between the nodes, it is not efficient. Besides, the traces can be used on several sub-images at the same time. This one-to-many dependency may not be properly understood by HPX, especially in the case where the data could be duplicated to run tasks on multiple instances of the data at the same time. In this case, the one-to-many dependencies between the trace and the sub-image could be transformed by HPX into a sequence of tasks processing sub-images one by one instead of processing them at the same time. It could restrict the execution flow of the tasks and reduce the performance.

MPI+OpenMP has a scaling close to MPI. However, it is less efficient than pure MPI parallelization. MPI has the best performances and also scales well. The applications
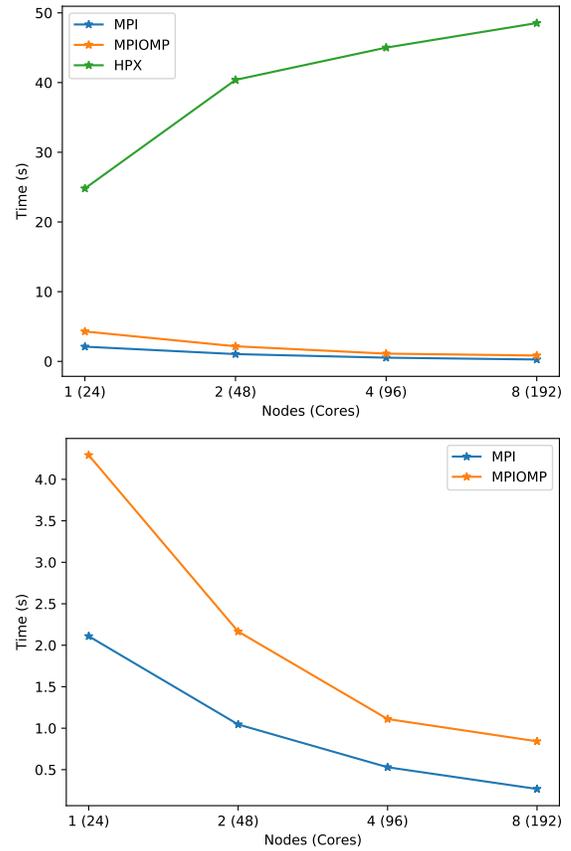

Fig. 4. Strong scaling considering HPX, MPI and MPI+OpenMP for a 15000 × 15000 points image on Pangea II. The second figure is zoomed on MPI and MPI+OpenMP.

use the same kernel but the management of the memory is up to the programming model used to implement the application. This explains the difference in performances between MPI and HPX. As for the differences between MPI and MPI+OpenMP, the OpenMP directives introduced in the kernel do not produce multi-threaded code as efficient as the multi-process code produced by the pure MPI parallelization for this application. This will be further discussed in Section IV-D.

C. Weak Scaling

Weak scaling experiments were performed on an image in which the number of points increases with the number of nodes used. The base image size is 15000 × 15000 and the first dimension is multiplied by the number of nodes. For instance, the image used for 4 nodes is 60000 × 15000. We executed our Kirchhoff seismic pre-stack depth migration MPI, MPI+OpenMP and HPX applications on 10 generated traces with propagation times. We study the performance improvement of the migration while the number of nodes (cores) and the size of the image increases.

Figure 5 shows the results of our weak scaling experiments on the Kirchhoff seismic pre-stack depth migration
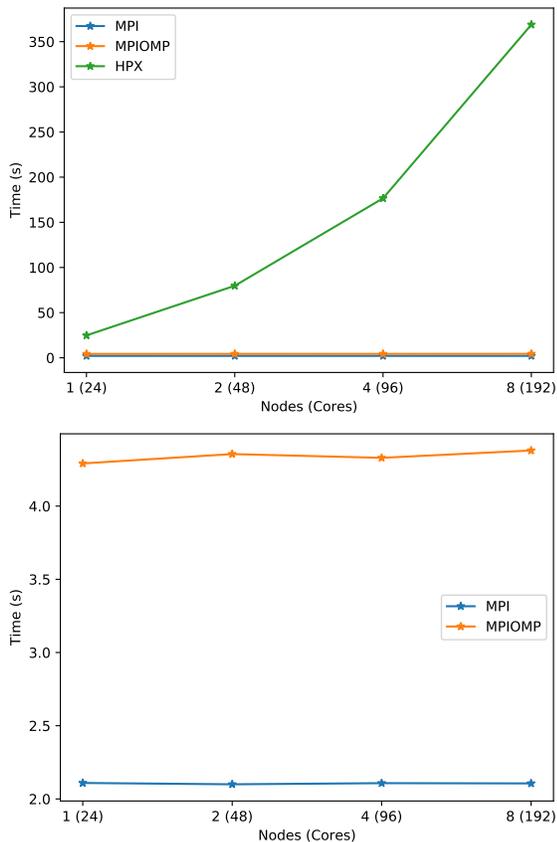
Fig. 5. Weak scaling considering HPX, MPI and MPI+OpenMP for a 15000 × 15000 points image on Pangea II. The second figure is zoomed on MPI and MPI+OpenMP.

for an image of base size of 15000 × 15000 which grows with the number of nodes (cores) and 10 traces. The ideal weak scaling is a constant execution time when the computing resources allocated to the application are increasing in the same proportions as the data. This is not the case for HPX. This may be due to HPX memory management and unexpected image and trace migrations between nodes which we are not able to control. The MPI and the MPI+OpenMP applications have a weak scaling almost perfect since the execution time stays almost constant while increasing the image size and the number of nodes in the same proportions.

D. Variation of the Number of OpenMP Threads

In this section, we will focus on the performances of our MPI+OpenMP implementation. Experimentations on the MPI+OpenMP Kirchhoff seismic pre-stack depth migration application with a different number of OpenMP threads allocated to the application have been performed for an image of 15000 × 15000 points. On one hand, we used only one process and changed the number of OpenMP threads allocated to the application until all the cores of the node are used. On the other hand, we used all the cores of the node but we changed the number of processes

allocated to the MPI+OpenMP as well as the number of OpenMP threads allocated per process to keep one thread per core. We test the performances of our migration kernel with the addition of the OpenMP directives.

Threads	Cores	Median execution time (s)
2	24	63.383
4	24	65.3781
6	24	71.5322
12	24	76.7288
24	24	254.1039

TABLE I
Execution time for a pure OpenMP application while increasing the number of OpenMP threads allocated to the application for a 15000 × 15000 point image on Pangea II.

Table I shows the execution times of the kernel depending on the number of threads allocated to the pure OpenMP application on one node of Pangea II. We can see that the execution time does not change much while increasing the number of threads except for 24 threads where it is very high. The execution time should decrease with the increase of parallel resources to solve the same problem. This shows that our OpenMP implementation of the migration kernel is not efficient enough to take advantage of the available cores.

Processes	Threads	Cores	Median execution time (s)
2	12	24	46.7809
4	6	24	6.3982
6	4	24	4.3045
12	2	24	8.7822
24	1	24	2.0901

TABLE II
Execution time for a hybrid MPI+OpenMP application while increasing the number of MPI process while keeping 24 OpenMP threads allocated to the application for a 15000 × 15000 point image.

Table II shows the execution times for the MPI+OpenMP executed on one node of Pangea II where all the cores run an OpenMP thread and the number of MPI processes changes. The number of threads allocated for each MPI process is the number of cores (24) divided by the number of MPI processes allocated to the application. We can see that the pure MPI application (24 processes and 1 thread per process) is the fastest case. There are also two unexpected values; for 4 processes with 6 threads per process and 12 processes with 2 threads per process. These values do not line up with the rest. Especially the value for 12 processes with 2 threads per process that should reasonably be between the value for 6 and 24 processes. The best performances are obtained with 6 processes and 4 OpenMP threads per process. These are the values used for our strong and weak scaling experiments with MPI+OpenMP since they obtain the best performances in both cases.

E. Discussions on performances

The migration kernel has to make interpolation of the wave travel time from the receiver and the source to

reach the considered point. This interpolation accesses the array containing the propagation times to perform the interpolation at the coordinates of the point. These accesses may be difficult to predict for the compiler thus the OpenMP runtime is not able to efficiently divide and parallelize the loops during the iteration over the image points while computing the interpolation of the travel times at the point. Thus, our implementation of the kernel may not be written in a suitable way for the compiler and the OpenMP runtime to efficiently parallelize it. Therefore, using a pure MPI implementation avoid this problem by splitting the image and the propagation times beforehand at the process level. Then, the kernel is executed on each sub-image without the necessity to try to efficiently parallelize the loops iterating over the points to perform the interpolations since they are already split before calling the kernel.

In conclusion of our scaling experiments on the implementations of the Kirchhoff seismic pre-stack depth migration, our HPX application does not scale very well both in terms of weak and strong scaling compared to our MPI application. In the MPI application, the location of each piece of data can be exactly controlled and there are no communications during the migration. On the contrary, the data migrations in our HPX application are up to the runtime of the programming model. Therefore, there may be migration of the images and the traces across the nodes which reduce the performances. Besides, the one-to-many dependencies between a trace and the sub-images may be expressed sequentially in HPX instead of in parallel which could restrict the execution and reduce the performances. Moreover, there is a conversion between the C data structures used in the C kernel and the C++ data structure used in HPX that may also decrease the performances of the application since the data are not converted in the C-based MPI application. Finally, the introduction of OpenMP directives in the kernel is not successful since the performances are not improved with a hybrid MPI+OpenMP implementation compared to a pure MPI implementation.

V. Conclusion

We introduced task-based algorithms for the Kirchhoff seismic pre-stack depth migration and the parallelism intrinsic to the method. Then, we introduced and implemented a simplified 2D version of the Kirchhoff seismic pre-stack depth migration as a task-based application with HPX. We also implemented MPI and MPI+OpenMP applications to compare with our task-based implementation. Then, we performed strong scaling and weak scaling experiments as well as studied the influence of the number of OpenMP threads on our kernel. We showed that our HPX application does not scale very well both in terms of weak and strong scaling compared to our MPI application. We deduced that there were migrations of the images and the traces across the nodes which reduced

the performances. We also showed that our addition of OpenMP directives in the kernel did not bring as good performances as our pure MPI application.

For the Kirchhoff seismic pre-stack depth migration, the fine-grain management of the memory, the careful alignment of the data, and the IOs allowed by the pure MPI implementation yields better performances than the higher-level implementations such as HPX and MPI+OpenMP which are not able to cope efficiently with the data placement, migrations, and dependencies. Indeed, for this kind of application which is highly parallel in term of computations and have irregular IOs, HPX and MPI+OpenMP are less suitable than MPI. In this case, a higher level of application description was not as efficient to manage the local data necessary for the migration.

Therefore, a solution to this issue is to improve how HPX manages the dependencies and make sure that it reduces the memory operations that are not necessary and costly. This could be achieved by pinning data to nodes and trying to avoid sending these data between nodes. In this case, a trace was used as input of the migration for multiple images, thus there is a one-to-many consumers operation and it is mandatory to improve how the scheduler handles such dependencies.

Task-based programming models are an interesting alternative to MPI due to their capacity to schedule computations and data migrations to reduce costly communications while optimizing the use of the computing resources. Therefore, the scheduler could anticipate data migrations and, in particular, IOs with the file system to load data in advance so that they are ready to be used. Moreover, it has been shown that task-based programming models can have better performances than regular MPI applications [12] [6] for dense linear algebra methods. However, in this case, we did not manage to implement a very efficient application due to several issues.

Acknowledgment

References

[1] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," IEEE Comput. Sci. Eng., vol. 5, no. 1, pp. 46–55, Jan. 1998. [Online]. Available: https://doi.org/10.1109/99.660313

[2] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," Queue, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[3] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. DeBardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen, "Addressing failures in exascale computing," The International Journal of High Performance Computing Applications, vol. 28, no. 2, pp. 129–173, 2014. [Online]. Available: https://doi.org/10.1177/1094342014522573

[4] O. E. Aaker, E. B. Raknes, and B. Arntsen, "Elastodynamic full waveform inversion on gpus with time-space tiling and wavefield reconstruction," The Journal of Supercomputing, vol. 77, no. 3, pp. 2416–2457, Mar 2021.

[5] L. Boillot, G. Bosilca, E. Agullo, and H. Calandra, "Task-based programming for seismic imaging: Preliminary results," in 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS), 2014, pp. 1259–1266.

[6] J. Gurhem and S. G. Petiton, "A current task-based programming paradigms analysis," in Computational Science – ICCS 2020, V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, and J. Teixeira, Eds. Cham: Springer International Publishing, 2020, pp. 203–216.

[7] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, and D. S. Nikolopoulos, "A taxonomy of task-based parallel programming technologies for high-performance computing," The Journal of Supercomputing, vol. 74, no. 4, pp. 1422–1434, Apr 2018. [Online]. Available: https://doi.org/10.1007/s11227-018-2238-4

[8] L. Ping and C. Yunhe, "Seismic 3d prestack time migration on parallel computers," Journal of Systems Engineering and Electronics, vol. 6, no. 3, pp. 49–55, 9 1995.

[9] J. Panetta, T. Teixeira, P. R. P. de Souza Filho, C. A. da Cunha Finho, D. Sotelo, F. M. R. da Motta, S. S. Pinheiro, I. P. Junior, A. L. R. Rosa, L. R. Monnerat, L. T. Carneiro, and C. H. B. de Albrecht, "Accelerating kirchhoff migration by cpu and gpu cooperation," in 2009 21st International Symposium on Computer Architecture and High Performance Computing, 10 2009, pp. 26–32.

[10] R. Xu, M. Hugues, H. Calandra, S. Chandrasekaran, and B. Chapman, "Accelerating kirchhoff migration on gpu using directives," in 2014 First Workshop on Accelerator Programming using Directives, 11 2014, pp. 37–46.

[11] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, ser. PGAS '14. New York, NY, USA: ACM, 2014, pp. 6:1–6:11. [Online]. Available: http://doi.acm.org/10.1145/2676870.2676883

[12] J. Gurhem, M. Tsuji, S. G. Petiton, and M. Sato, "Distributed and parallel programming paradigms on the k computer and a cluster," in Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, ser. HPC Asia 2019. New York, NY, USA: ACM, 2019, pp. 9–17. [Online]. Available: http://doi.acm.org/10.1145/3293320.3293330